Robert Andrei Buchmann
Ana-Maria Ghiran

Tehnologii semantice

Presa Universitară Clujeană

Robert Andrei Buchmann Ana-Maria Ghiran

TEHNOLOGII SEMANTICE

Presa Universitară Clujeană

2018

Această carte a fost realizată și publicată cu sprijinul proiectului de cercetare finanțat prin programul UEFISCDI PN-III-P2-2.1-PED-2016-1140 (ENTERKNOW) coordonat de Prof. Dr. Robert Andrei Buchmann.

Referenți științifici:

Prof. dr. Nicolae Tomai

Prof. dr. Gheorghe Cosmin Silaghi

ISBN 978-606-37-0374-4

Universitatea Babeş-Bolyai
Presa Universitară Clujeană
Director: Codruța Săcelean
Str. Hasdeu nr. 51
400371 Cluj-Napoca, România
Tel./Fax: (+40)-264-597.401
E-mail: editura@editura.ubbcluj.ro
http://www.editura.ubbcluj.ro/

Cuprins

Introducere 1

1. Tehnologii bazate pe XML 3

1.1 Interfața Oxygen și buna formare XML 3

1.2 Validarea cu vocabulare XML Schema 16

1.3 Transformări XSLT 26

2. Interoperabilitate sintactică bazată pe JSON și cereri AJAX cross-domain 42

2.1 Crearea de domenii virtuale multiple în aceeaşi instalare Apache 42

2.2 Crearea de cereri HTTP prin XMLHttpRequest 44

2.3 Crearea de cereri HTTP cross-domain prin tehnica JSON-P 45

2.4 Crearea de cereri HTTP cross-domain prin scripturi server proxy 49

2.5 Crearea de cereri HTTP cross-domain prin tehnica CORS 54

2.6 Crearea de cereri spre servere JSON 60

2.7 Cereri asincrone în scripturi proxy PHP 66

3. Reprezentarea cunoștințelor în grafuri RDF 69

3.1 Comparaţii cu XML 69

3.2 Sintaxele Turtle și N-triples 69

3.3 Afirmații cu valori literale 71

3.4 Afirmații cu noduri anonime 72

3.5 Conversii sintactice 72

3.6 Vocabulare RDF 75

3.7 Gruparea afirmațiilor în mai multe grafuri 78

4. Interogarea grafurilor de cunoștințe prin SPARQL 79

4.1 Interogări SPARQL in RDF4J 79

4.2 Interogări SPARQL complexe 87

4.3 Interogări SPARQL Update în RDF4J 100

4.4 Interogări la distanță 105

5. Procesarea grafurilor RDF în PHP 112

6. Reprezentarea și procesarea grafurilor prin cod HTML distilabil 124

6.1 Sintaxa JSON-LD 124

6.2 Cerințele Google privind grafuri încorporate 126

6.3 Sintaxa HTML microdata și terminologia Schema.org 128

6.4 Distilarea microdata cu JavaScript 133

6.5 Crearea unei hărți GoogleMaps 136

6.6 Marcarea Google Maps pe baza microdatelor 140
6.7 Descrierea grafurilor cu JSON-LD 143
6.8 Interogarea Google Knowledge Graph 147
6.9 Distilarea grafurilor RDF în Python 149
7. Inferențe RDF Schema 171
8. Crearea unui instrument de modelare conceptuală agilă 178
8.1 Implementarea unui limbaj agil de modelare 180
8.2 Elemente de sintaxă 188
8.3 Elemente de semantică 191
8.4 Notație dinamică 195
8.5 Extinderea limbajului de modelare 198
8.6 Interogarea modelelor 204
8.7 Atribute specializate 206
8.8 Integrarea modelelor conceptuale cu grafuri RDF 226
8.9 Structura grafurilor exportate 241
Documentația standardelor utilizate 250
Bibliografie științifică 251

Introducere

Volumul de față oferă un conținut cu caracter practic, menit să familiarizeze cititorii cu tehnologiile de procesare și interoperabilitate semantică, incluzând însă și unele capitole despre interoperabilitatea sintactică care asigură fundația pentru prima categorie. Este așadar vorba despre cele trei formate de interoperabilitate promovate în cadrul așa-numitei generații de aplicații Web 3.0, și anume XML, JSON și RDF.

Acestora li se adaugă modelarea conceptuală care se încadrează în categoria tehnologiilor semantice, în măsura în care e utilizată în scopul reprezentării cunoștințelor (având și o manifestare grafică, diagramatică). Includerea acestora derivă din propunerea de a asigura interoperabilitate între modele conceptuale și bazele de date semantice în format RDF, propunere menită să permită realizarea unui nou tip de aplicații Web de tip model-driven - idee centrală a proiectului cu sprijinul căruia a fost realizată această carte - UEFISCDI PN-III-P2-2.1-PED-2016-1140[1].

Prin tutorialele și exercițiile prezentate aici ne adresăm atât studenților care doresc să se familiarizeze cu tehnologiile menționate, cât și tinerilor cercetători ce doresc să se inițieze în utilizarea acestora printr-o abordare practică, bazată pe exemple intuitive de complexitate scăzută, care să poată fi înțelese fără o aprofundare prealabilă a bazelor teoretice.

Unele concepte teoretice sunt explicate în introducerile capitolelor respective sau în paralel cu desfășurarea exercițiilor, însă doar în limita necesară înțelegerii modului de construire și rațiunii care stă la baza exemplelor. Am încercat să distilăm și să oferim doar esența explicațiilor teoretice, care adesea constituie o piedică pentru faza de inițiere în lucrul cu tehnologii semantice, datorită abstractizărilor formale pe baza cărora au fost construite. Pentru partea de fundamentare teoretică, standardele utilizate oferă on-line specificații oficiale ce pot fi consultate liber și nu vor fi reproduse în această carte, fiind doar referite în bibliografie. Evităm, pe această cale, să oferim o carte care se limitează la a compila conținuturi deja accesibile și urmărim să încurajăm novicii în adoptarea acestor tehnologii pentru probleme concrete, la nivel de utilizare și programare.

În cele ce urmează prezentăm pe scurt structura cărții și modul de aliniere conceptuală a capitolelor:

XML a jucat un rol important în redefinirea noțiunii de interoperabilitate, oferind un set de reguli în ceea ce privește formatul în care trebuie exprimate datele transmise între aplicațiile software. Capitolul 1 prezintă modul de interogare a structurilor XML, prezintă validarea documentelor XML în raport cu un anumit vocabular și tehnici de transformare a documentelor XML în alte documente care să corespundă cerințelor de input a sistemelor de procesare a acestora. O alternativă la XML este formatul JSON mai performant ca structură de date ce trebuie transferată între aplicații. Capitolul 2 analizează interoperabilitatea sintactică în format JSON, evidențiind trei modalități de realizare a cererilor HTTP între domenii diferite: JSON-P, scripturi server proxy, și CORS. Totodată sunt prezentate și bazele de date bazate JSON și mecanisme de interogare a acestora prin HTTP.

Capitolul 3 exemplifică utilizarea formatului RDF pentru baze de date semantice sau baze de cunoștințe. RDF permite mai multe sintaxe de serializare pentru structuri de tip graf, iar acest capitol prezintă câteva dintre acestea: Turtle, N-Triples, TriG. RDF folosește propriul limbaj de interogare, adecvat modului de reprezentare a datelor prin grafuri - SPARQL. Capitolul 4 prezintă interogări în limbajul SPARQL pornind de la extragerea de cunoștințe locale, până la agregarea unor rezultate din surse de date externe folosind interogări federative. Capitolul 5 exemplifică procesarea grafurilor folosind biblioteca EasyRDF pentru limbajul PHP. Capitolul 6 tratează reprezentarea cunoștințelor în cod HTML iar pentru aceasta sunt prezentate sintaxele JSON-LD, microdata, RDFa. Pentru procesarea datelor, pe lângă biblioteca EasyRDF,

[1] http://enterknow.granturi.ubbcluj.ro

prezentată în capitolul anterior, exemplificăm și alte biblioteci - RdfLib, SPARQLWrapper – ale limbajului Python. Valoarea bazelor de cunoștințe este amplificată când pe lângă extragerea de informații existente, se pot deduce cunoștințe noi. Capitolul 7 prezintă posibilitățile de inferență încorporarea unor reguli și axiome.

Pe lângă reprezentarea formală a cunoștințelor dată de tehnologiile Semantic Web, putem întâlni o reprezentare a acestora și sub forma modelelor conceptuale folosite pentru a exprima cunoștințele experților sub forma unor diagrame utilizând un limbaj de modelare. Dacă în mod uzual modelarea urmează indicațiile impuse de diverse limbaje standardizate, solicitând modelatorului cunoașterea standardului ales, în prezent există tendințe de a facilita accesul celor care doresc să își personalizeze, extindă sau simplifice limbajul de modelare în conformitate cu nevoile proprii sau cu specificiul unui domeniu de aplicare. Astfel de limbaje pot fi cuplate cu tehnologia RDF pentru a expune diagramele într-un format interogabil pe baza căruia să se poată construi aplicații de tip model-driven, beneficiind de tehnologii semantice. Astfel, ultimul capitol al acestui volum se va concentra pe definirea unui limbaj de modelare ce permite exprimarea de cunoștințe într-o formă diagramatică folosind concepte specifice unui domeniu de interes. Exemplul prezentat folosește platforma ADOxx pentru prototipizarea rapidă a unui instrument de modelare. În acest scop, se apelează la metodologia de dezvoltare agilă a unor metode de modelare (AMME) ce facilitează dezvoltarea incrementală și ghidată de cerințe de modelare specifice.

În final este prezentat un exemplu demonstrativ prin care sunt extrase cunoștințe din modele diagramatice sub forma unor afirmații RDF, combinate cu date provenite din alte surse și în cele din urmă folosite la generarea unor noi cunoștințe. O versiune preliminară simplificată a exemplului a fost inițial publicată în teza de abilitare a Prof. Robert Buchmann[2], exemplul fiind extins aici cu toate detaliile necesare prezentării sale ca exercițiu.

[2] "The Semantic Enrichment of Linked Data Graphs with Knowledge Represented as Domain-Specific Diagrammatic Models", prezentată la Univ. Al. I. Cuza Iași în 2016

1. Tehnologii bazate pe XML

XML[3] este fundația Web 3.0, asigurând componenta de interoperabilitate sintactică. Mai exact, oferă un mod de a structura informație (date, text sau combinații între acestea) cu ajutorul unor reguli standard de delimitare numite **reguli de bună formare**. Documentele XML astfel structurate pot îndeplini multiple roluri: baze de date XML (caz în care punem accent pe validarea și interogarea lor), documente menite a fi citite de utilizatori umani (caz în care punem accent pe transformarea și stilizarea lor) sau structuri de date menite a fi transmise între aplicații diferite. Aplicațiile respective pot fi scrise în orice limbaj de programare deoarece statutul de standard al XML face ca acesta să fie procesat în moduri similare în orice mediu de programare.

Oxygen[4] este un instrument ușor de folosit pentru familiarizarea cu tehnologiile XML. Îl vom folosi pentru a oferi o introducere bazată pe exerciții de tip tutorial cu privire la (i) interogări XML cu ajutorul limbajului Xpath[5], (ii) impunerea unei structuri asupra documentelor XML și validarea acelei structuri cu ajutorul limbajului XML Schema[6], (iii) transformări XML cu ajutorul limbajului XSLT[7]. Explicațiile teoretice vor însoți exercițiile practice acolo unde e nevoie.

1.1 Interfața Oxygen și buna formare XML

Creați în Oxygen un document XML nou.

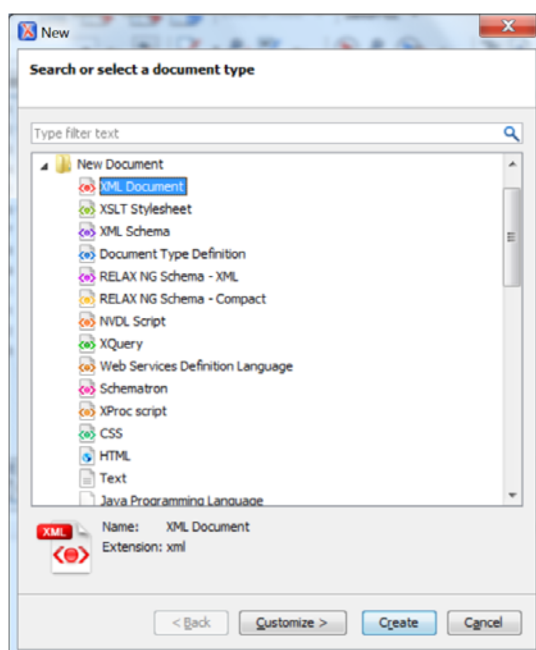


Figura 1 Creare document nou în Oxygen XML

În fereastra de creare se pot observa jos două butoane:

- butonul Customize ne permite să precizăm un vocabular, DACĂ dorim ca documentul nostru să se alinieze (să se valideze) față de regulile impuse de un vocabular;
- deocamdată nu creăm vocabulare, așa că apăsăm Create.

Introduceți următorul document XML:

[3] https://www.w3.org/XML/
[4] Disponibil la https://www.oxygenxml.com/
[5] https://www.w3.org/TR/xpath/
[6] https://www.w3.org/standards/xml/schema
[7] https://www.w3.org/TR/xslt/

```
<?xml version="1.0" encoding="UTF-8"?>
<produs pret="100">Televizor</produs>
<produs pret="200">Ipod</produs>
<pret total>300</pret total>
```

Imediat ce tastați, vi se atrage atenția că acesta **nu e un document XML bine format**. Acest mesaj apare automat în <u>partea de jos</u> a documentului sau, dacă apăsați <u>butonul de testare</u> a bunei formări, în **<u>zona de output</u>**.

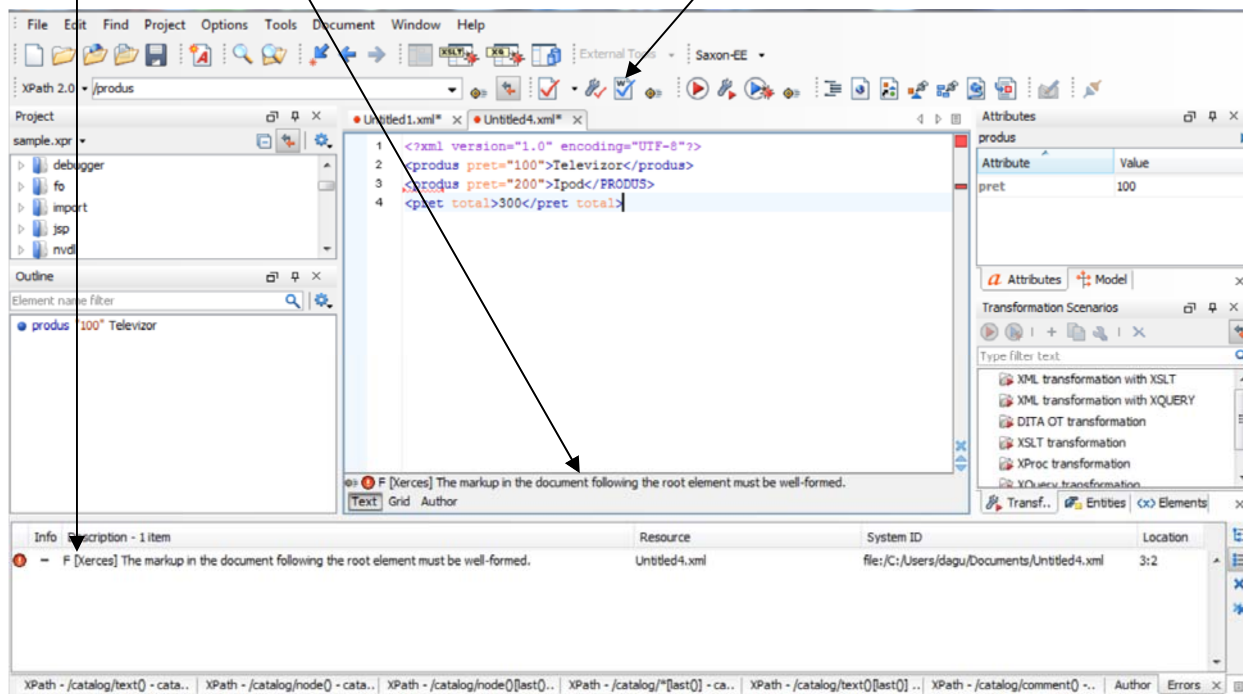


Figura 2 Interfața Oxygen XML

Identificați regulile de bună formare pe care le încalcă documentul:

- documentul nu are rădăcină unică (are 3 elemente pe primul nivel);
- al doilea element nu se închide (</PRODUS> cu </produs> nu e totuna, deoarece XML e case sensitive)[8];
- al treilea element nu are voie să conțină spațiul, acesta fiind rezervat pentru separarea de atribute; practic se consideră că PRET e marcatorul iar TOTAL e un atribut lipsit de valoare, deci eronat (orice atribut trebuie să aibă o valoare între ghilimele).

În continuare creați următorul document bine format:

```
<?xml version="1.0" encoding="UTF-8"?>
<comanda>
  <!-- acesta e un comentariu -->
  <produs denumire="Televizor" id="p1" pret="100"/>
  <produs id="p2" pret="200">Calculator</produs>
  <produs>
    <id>p3</id>
    <pret>300</pret>
    <denumire>Ipod</denumire>
  </produs>
  <prettotal>600</prettotal>
  Aceasta este o comanda de produse
</comanda>
```

[8] În text se vor folosi majuscule de câte ori este vorba de marcatori și atributele lor, ca să fie mai ușor de citit textul.

Acesta e un document cu o structură nerecomandată, neregulată, dar ne permite să exersăm diverse aspecte.

Dați un click pe primul produs. Veți remarca o serie de aspecte printre care: generarea automată a unei căi Xpath absolute care **ar putea returna** elementul selectat; sublinierea cu gri a elementului selectat este importantă pentru căile RELATIVE, care vor fi calculate de la nodul subliniat; în dreapta se mai poate vedea și zona de gestiune a atributelor elementului selectat

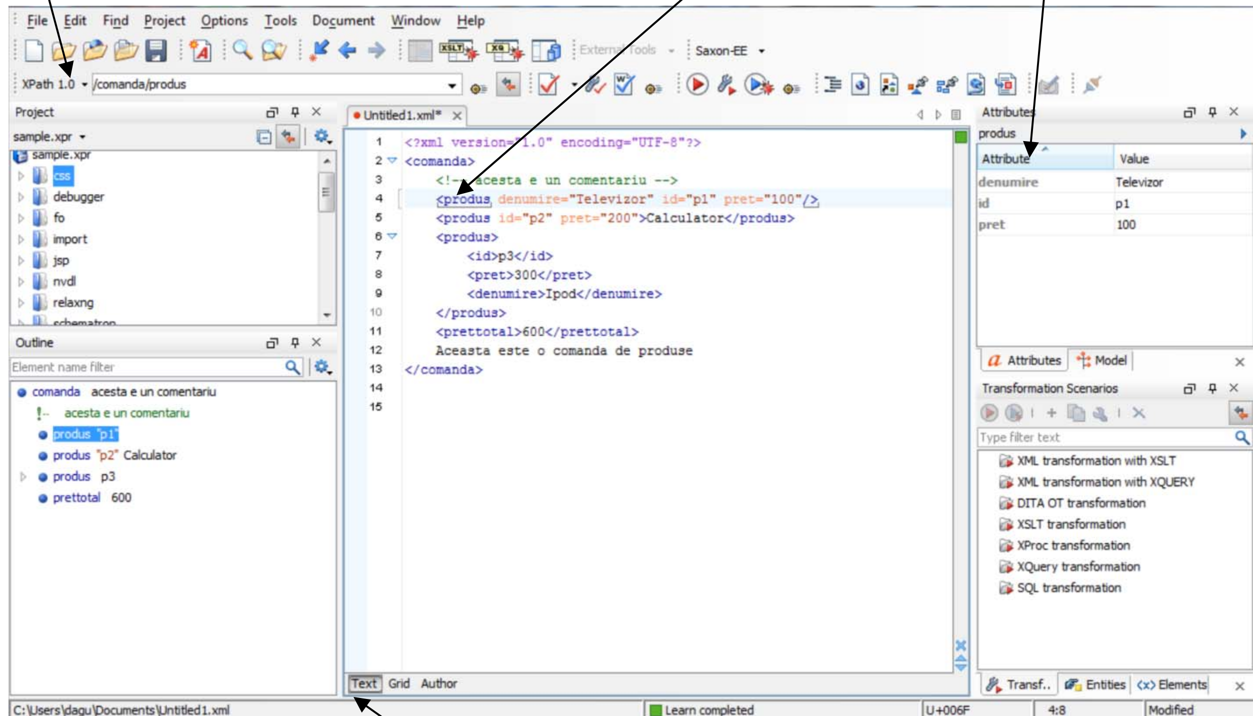


Figura 3 Interfața Oxygen XML pentru interogările Xpath într-un document XML

Alt aspect de interfață important e modul de vizualizare a documentului:

- modul Text este ceea ce se vede în acest moment;
- modul Author este varianta formatată, cu condiția să se fi creat stiluri CSS pentru marcatori;
- modul Grid este un alt mod de afișare a arborelui DOM, util pentru că ne arată exact câți copii are fiecare nod (în timp ce în modul Text acest lucru nu e întotdeauna evident).

De exemplu, la întrebarea câți copii are nodul COMANDA?, ați putea fi tentați să răspundeți cu 6:

- 1 comentariu, 3 elemente PRODUS, 1 element PRETTOTAL si 1 nod text.

În realitate numărul de copii este 11, pentru că mai există încă 5 noduri invizibile de tip text (apăsarea tastei Enter, însoțită uneori de spații sau Tab pt indentare!):

```
<?xml version="1.0" encoding="UTF-8"?>
<comanda>
  <!-- acesta e un comentariu -->
  <produs denumire="Televizor" id="p1" pret="100"/>
  <produs id="p2" pret="200">Calculator</produs>
  <produs>
    <id>p3</id>
    <pret>300</pret>
    <denumire>Ipod</denumire>
  </produs>
  <prettotal>600</prettotal>
  Aceasta este o comanda de produse
</comanda>
```

De ce nu sunt considerate noduri text și ultimele 5 treceri la rând nou?

- Primele trei sunt noduri text, dar sunt copiii celui de-al treilea PRODUS, nu al lui COMANDA;
- Ultimele două sunt ale lui COMANDA, dar fac parte din același nod text ca propoziția pe care o încadrează. Deci ultimul nod text este de fapt *{ENTER+TAB}Aceasta este o comanda de produse{ENTER}*

Prezența nodurilor invizibile poate afecta rezultatele interogărilor Xpath bazate pe poziție. În programare se evită în general prezența acestor noduri prin câteva metode:

- Dacă documentul XML e luat dintr-un fișier sau string, avem grijă ca în acel fișier/string să scriem totul pe un singur rând, fără Enteruri (dezavantaj: dificultate de citire și editare!)
- Se creează o funcție de curățare recursivă a nodurilor invizibile;
- Dacă se construiește documentul XML direct prin program, nod cu nod, această problemă nu mai apare deoarece funcțiile de lipire a nodurilor XML (appendChild, insertChild etc.) generează un XML curat, fără a introduce noduri invizibile! Acestea apar datorită tastării și aranjării codului XML pentru un cititor uman.

În Oxygen vom păstra Enterurile pentru citirea mai ușoară a exemplelor. Am amintit deja că vizualizarea în mod Grid ne evidențiază mai bine aceste **noduri invizibile**.

Afișarea în Grid se face pe nivele, cu posibilitatea de a detalia sau restrânge conținutul fiecărui element, folosind <u>săgețile.</u> Printre elemente se pot vedea nodurile invizibile, cu <u>numele</u> și <u>valoarea (rând gol).</u>

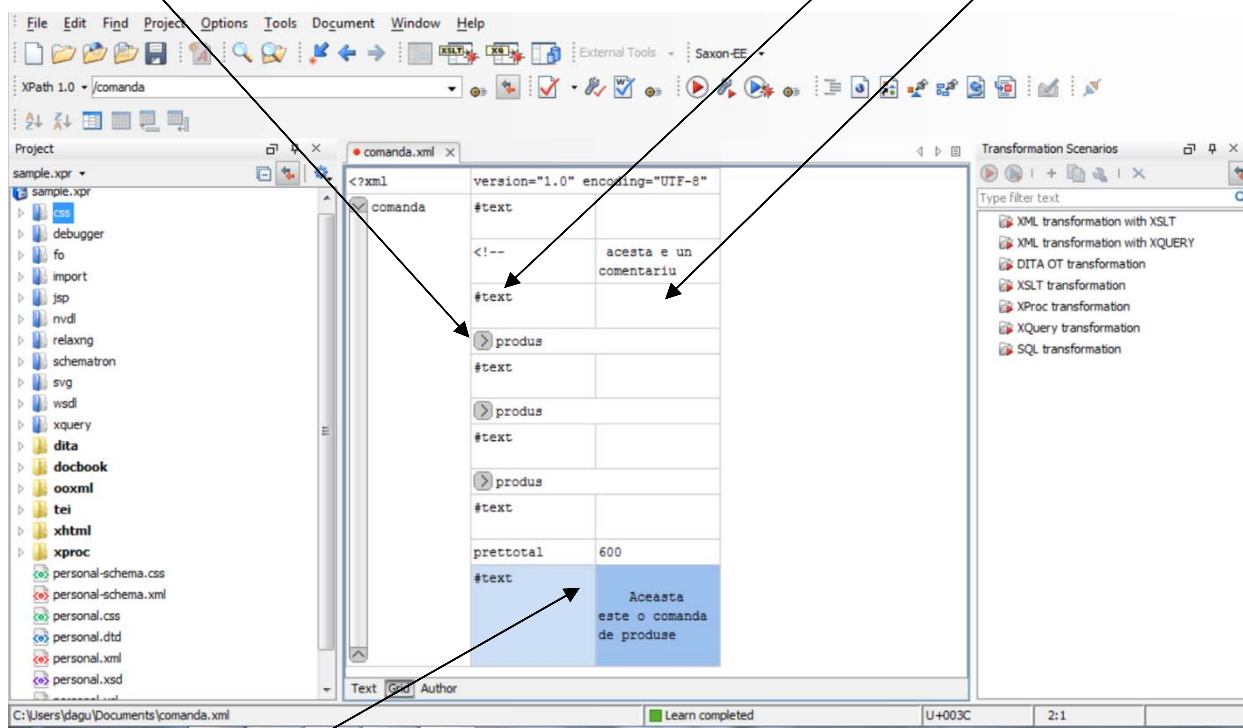


Figura 4 Arborele DOM în modul de vizualizare Grid

La ultimul nod text se poate vedea că a fost reunit cu un Enter (rând gol) și un Tab în față, iar în spate cu un Enter (comparativ, valoarea 600 de deasupra, nu are nici un rând gol sau spațiere).

Într-o vizualizare abstractă, arborele DOM al acestui exemplu arată astfel:

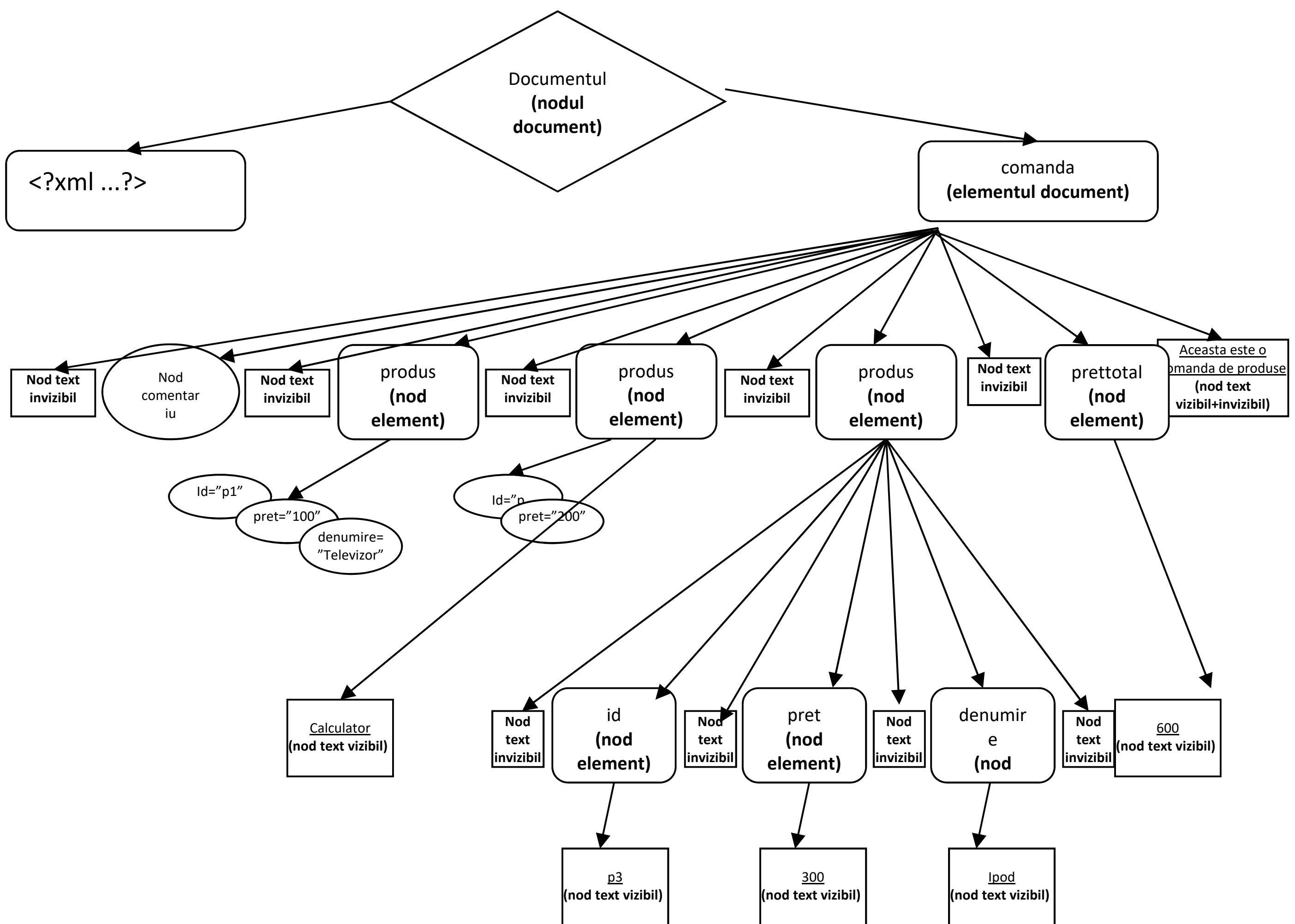


Figura 5 Vizualizarea grafică a arborelui DOM

Acest arbore are 5 nivele (atributele nu formează un nivel!)

Există câteva dileme cu care se confruntă începătorii (datorită unor confuzii legate de terminologie, confuzii generate de literatura de specialitate):

Confuzia 1: Rădăcina

Termenul **Rădăcină** este folosit cu două sensuri:

- pentru a se referi la rădăcina reală a arborelui DOM (documentul în ansamblu, termenul standard fiind *the document node*, **nodul document**);
- pentru a se referi la marcatorul/elementul rădăcină (COMANDA în cazul nostru, termenul standard fiind *the document element*, **elementul document**).

Autorii folosesc în mod liber cuvântul **rădăcină** pentru a se referi când la unul, când la celălalt. *În general, vom folosi termenul **rădăcină** pentru a referi marcatorul care conține toți ceilalți marcatori (deci elementul document), considerând că toată lumea știe că acesta trebuie obligatoriu precedat de declarația <?xml...?>.*

Confuzia vă poate afecta cel mai des atunci când creați un arbore DOM prin programare, nod cu nod. Există tentația de a crea toate elementele, a le lipi între ele, dar a uita că și elementul rădăcină trebuie lipit la document.

De exemplu, în PHP, crearea unui arbore DOM trebuie să înceapă cu linia:
$doc=new DOMDocument();
(crearea nodului document)
....și să se termine cu:
$doc->appendChild($radacina)
(lipirea rădăcinii la document, după ce în rădăcină s-a introdus tot ce trebuie)

Confuzia 2: Atributele sunt noduri?

Această confuzie e legată de semnificația atribuită termenilor **nod** și **atribut**, confuzie creată chiar de standardul DOM, care se referă la atribute prin expresia "noduri de tip atribut".

Totuși, în general **atributele sunt tratate diferit de noduri**. De exemplu:

- interogările Xpath după funcția **node()** vor returna ORICE nod (de orice tip: text, elemente, comentarii etc.) dar nu și atribute (căutarea generică de atribute se face cu **@***);
- nodurile au o ordine relevantă (le accesăm de obicei pe bază de poziție) în timp ce atributele de obicei se accesează pe bază de nume (ordinea lor e irelevantă);
- atributele vor fi ignorate de funcțiile care caută pe baza relațiilor de rudenie (copil/frate/etc.); chiar și în standardul DOM, atributele nu fac parte din vectorul childNodes, au propriul vector, numit attributes.

Cel mai sigur este să vă gândiți la atribute ca la niște **proprietăți ale nodurilor de tip element** și nu ca la noduri-copil ale acestora. Din acest motiv în desenarea arborelui DOM nu am desenat atributele pe următorul nivel (4), ci între nivele (3 și 4) sugerând că au un statut aparte și nu sunt noduri copil.

Confuzia 3: Numele și valoarea nodurilor

O confuzie care afectează mai mult la nivel de programare este semnificația termenilor **valoare de nod**/ **nume de nod**.

Conform standardului DOM, orice nod (și orice atribut) trebuie să aibă **nume** și **valoare**. În cazul atributelor acest lucru e evident, dar la elemente și noduri text e mai puțin evident. De exemplu care e numele și valoarea în cazul <produs>Televizor</produs> ?

Răspunsul intuitiv ar fi că “produs” este numele și “televizor” valoarea. În realitate, există două viziuni:

A. Standardul DOM specifică astfel:
În acest exemplu avem 2 noduri, unul de tip element, cu un nod copil de tip text. FIECARE din acestea are un nume și o valoare!

- Nodul element are numele “produs” și valoarea null!
- Nodul text are numele standard #text și valoarea “Televizor”!

Orice programator care lucrează cu standardul DOM va trebui să țină cont de acest fapt (stringul "Televizor" nu e valoarea directă produsului, ci valoarea primului copil al nodului PRODUS). Pentru accesarea stringului Televizor, prin funcții DOM va fi necesară o construcție de genul (PHP):

$x=..... (stocarea elementului PRODUS prin getElementBy…, childNodes[] sau alte metode)
$text=$x->firstChild->nodeValue (preluarea textului din marcator)
$tag=$x->nodeName (preluarea marcatorului)

B. Deși standardul DOM și funcțiile aferente tratează arborele DOM în acest mod, limbajul de interogare Xpath tratează nodurile în modul mai intuitiv/simplificat: „produs” este numele, „Televizor” este valoarea.

Xpath este limbajul fundamental de interogare a documentelor XML, indiferent de scopul lor. Pe Xpath se bazează majoritatea limbajelor care manipulează cod XML. Spre deosebire de SQL, Xpath poate doar citi informații, nu și modifica/insera/șterge. Pentru acestea se apelează la funcțiile standard DOM (dacă modificările se fac direct pe documentul inițial) sau la limbajul de transformare XSLT (dacă dorim să generăm un nou document în urma modificărilor).

Interogările Xpath se execută în caseta dedicată din stânga sus, unde se poate selecta și versiunea Xpath în care dorim să lucrăm. În timpul tastării unei interogări, apar liste de sugestii și un Help contextual, foarte util în familiarizarea cu cuvintele cheie din Xpath

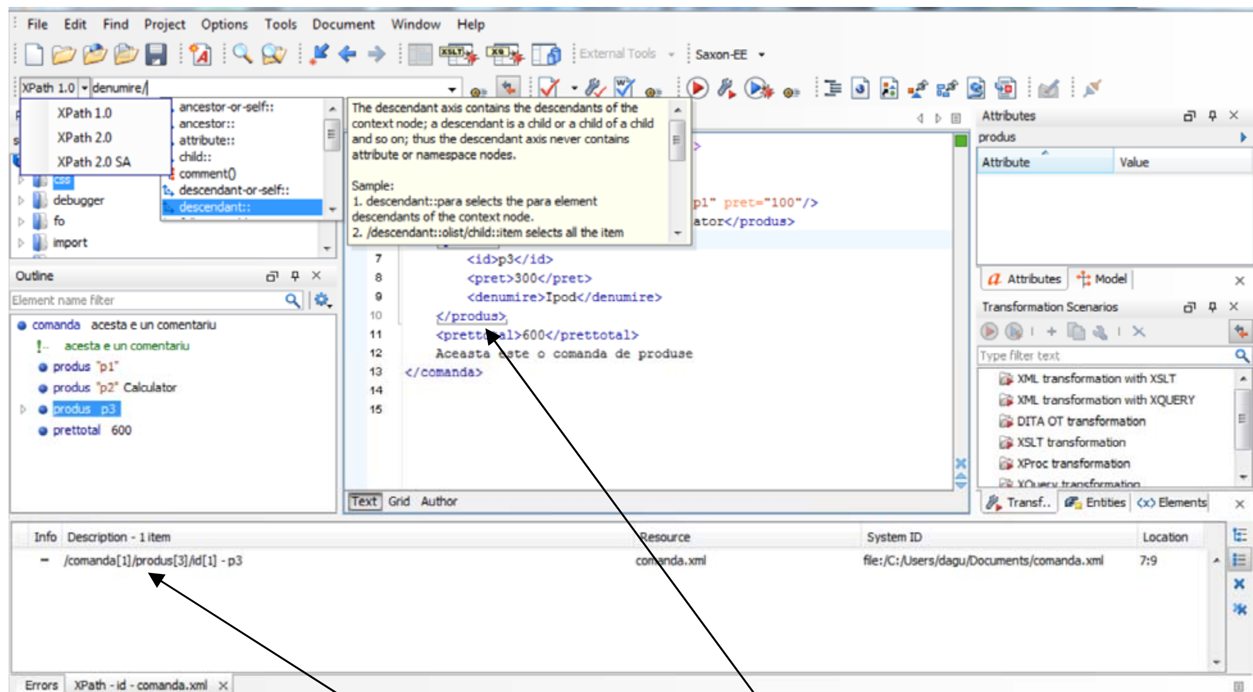


Figura 6 Interogările Xpath pe un document XML

Rezultatele se afișează în panoul de output:

În mod implicit, interogările sunt **căi absolute**, încep de la rădăcină. Dacă dorim **căi relative**, trebuie să alegem **nodul curent**, printr-un click pe marcatorul dorit, care va fi subliniat cu gri.

Executarea unei interogări nu dă rezultatul ca un simplu string, ci indică și poziția din document la care s-a găsit rezultatul:

/comanda/produs/id

....caută elementele ID, din PRODUS, din COMANDA și afișează rezultatul:

/comanda[1]/produs[3]/id[1] – p3

....indică faptul că rezultatul se află în primul COMANDA, al treilea PRODUS, primul ID și are valoarea “p3”.

Exersați interogările indicate în continuare:

Căi absolute simple (fără axe, fără condiții)

/comanda/produs/@id

...caută atributul ID al elementelor PRODUS din COMANDA (se vor găsit 2 soluții, cu valorile „p1” și “p2”)

/comanda/produs

....toate elementele PRODUS din COMANDA (3 soluții)

/comanda
...toate elementele COMANDA (returnează elementul rădăcină)

/
... returnează nodul document

Căi generice

/comanda/*
...toate elementele din COMANDA (4 soluții)

/comanda/text()
....toate nodurile text din COMANDA (6 soluții, din care 5 sunt nodurile invizibile)

/comanda/node()
....toate nodurile din COMANDA, indiferent de tip (11 soluții din care 6 text, 4 element, 1 comentariu)

/comanda/comment()
....toate comentariile din COMANDA (1 soluție)

/comanda/produs/@*
....toate atributele din toate elementele PRODUS din COMANDA (5 soluții, 3 la primul PRODUS, 2 la al doilea)

/comanda/produs/node()
...toate nodurile din toate elementele PRODUS din COMANDA (*se observă că nu s-au returnat atributele!* 8 soluții, din care la al doilea PRODUS un text, iar la al treilea 3 elemente și 4 noduri invizibile)

Dacă dorim și nodurile și atributele scriem două interogări separate prin bară verticală:
/comanda/produs/node()|/comanda/produs/@*

În acest fel, Xpath 1.0 permite **reunirea soluțiilor din mai multe căi**. Xpath 2.0 oferă și tehnici mai avansate (intersecție, diferență etc.)

Căi cu rezultat boolean

Acestea nu returnează noduri sau valori din document, ci verifică dacă există sau nu anumite noduri sau valori:

/comanda/produs/id='p3'
...există nodul ID cu conținutul (valoarea) p3 în vreunul din elementele PRODUS din COMANDA? (true)

/comanda/produs/pret>100
....există vreun PRET cu valoare peste 100 în vreun PRODUS din COMANDA? (true)

/comanda/*=600
...există vreun element cu valoarea 600 în COMANDA? (true, e PRETTOTAL)

/comanda/*/pret=300
...are COMANDA un nepot PRET cu valoarea 300? (true)

/comanda/*/@*="Televizor"
....există vreun atribut cu valoarea “Televizor” în oricare din fiii lui COMANDA? (true)

/*/*="Calculator"
...are elementul rădăcină un fiu al cărui conținut e stringul “Calculator”? (true)

/*/*/*="Ipod"
...are elementul rădăcină un nepot cu conținutul “Ipod”? (true)

Ca regulă generală, interogările booleene se termină cu o expresie logică care NU apare între paranteze pătrate (dacă apare, înseamnă că e o restricție aplicată nodului precedent, vezi mai jos).

Căi de căutare

Când nu suntem siguri pe care ramură și pe ce nivel se găsește rezultatul se pot realiza căutări. În general nu sunt recomandate datorită performanțelor slabe pe arbori DOM foarte mari (trebuie parcurse toate ramurile descendente!). Căutarea este indicată prin //

//@*=100
...există oriunde în document un atribut cu valoarea 100? (true)

//*="Ipod"
...există oriunde în document un element cu valoarea Ipod? (true)

/comanda//@id="p1"
...există oriunde în COMANDA un atribut ID cu valoarea p1? (true)

/comanda//@id
...returnează toate atributele ID din COMANDA, indiferent de nivelul pe care se află (2 soluții)

//@id|//id
...returnează toate IDurile din document, indiferent că sunt atribute sau elemente, indiferent cui aparțin (3 soluții)

Căi cu restricție (condiție)

Restricția se pune între [] și poate fi aplicată oricărui nod din cale!
Cele mai simple sunt restricțiile de poziție:

/comanda/produs[1]
...returnează primul PRODUS din COMANDA (atenție, în Xpath-ul din Oxygen numerotarea începe de la 1, dar în unele browsere poate începe de la zero!)

/comanda/produs[last()]
...returnează ultimul PRODUS din COMANDA

//@*[1]
...returnează toate atributele ce apar primele în elementele de care aparțin (2 soluții)

//*[last()]
...returnează toate elementele ce apar ultimele în cadrul nodului de care aparțin (3 soluții, inclusiv rădăcina, care e ultima din document!)

Sunt frecvente și retricțiile privind conținutul:

/comanda/produs[id]
...returnează acele PRODUSE din COMANDA care conțin elemente ID (1 soluție)

/comanda/produs[@id]
...returnează acele PRODUSE din COMANDA care conțin atribute ID (2 soluții)

/comanda/produs[text()]
...returnează acele PRODUSE din COMANDA care conțin noduri text (2 soluții)

//*[text()="Calculator"]
...returnează toate elementele care conțin nodul text “Calculator” (1 soluție)

Restricțiile permit folosirea lui or, not, and pentru condiții mai complexe:

/comanda/produs[@id or id]
....returnează acele PRODUSE din COMANDA care au ID fie ca atribut, fie ca element (3 soluții; ne scutește de a scrie două căi separate prin |)

/comanda/produs[not(@id)]
...returnează acele PRODUSE din COMANDA care nu au atribut ID (1 soluție)

//produs[not(node())]
...returnează acel PRODUS care este vid (1 soluție)

//produs[not(@*)]
...returnează acel PRODUS care nu are atribute (1 soluție)

/comanda/produs[position()=1 and @id]
...returnează acel PRODUS care ocupă prima poziție (între frați) și are atribut ID (1 soluție)
(când combinăm restricția de poziție cu altele nu putem folosi produs[1 and @id], deoarece în expresii booleene orice număr diferit de zero e convertit în true, de aceea apelăm la funcția position)

Restricțiile nu se aplică doar pe ultima treaptă din cale. Putem să le aplicăm și undeva la mijlocul căii:

/comanda/produs**[@id]**/text()
...returnează nodurile text din acele PRODUSE cu atribut ID, din COMANDA (1 soluție)

Căi cu axe

Axele sunt direcțiile pe care înaintează calea. În mod implicit, o cale merge în jos. În mod explicit putem schimba direcția căii cu:

- @ pentru a merge pe axa atributelor;
- // pentru a căuta în nodul curent și toții descendenții săi;
- .. pentru a merge în sus, la părinte;
- . pentru a face referire la nodul curent (**folosit pentru a pune condiții asupra nodului curent**).

În general .. se folosește în căile relative, ca la foldere, pentru a urca un nivel în sus.

//text()/..
...returnează părinții tuturor nodurilor text (7 soluții)

//id/ancestor::node()
...returnează toți strămoșii elementelor ID (3 soluții)

//text()[.=”Calculator”]

...returnează toate nodurile text egale cu “Calculator” (1 soluție)

Vedeți cum s-a pus condiția asupra nodului curent cu ajutorul punctului!

- Dacă am fi scris //text()=”Calculator” am fi obținut o interogare booleană – există noduri text egale cu “Calculator”?
- Dacă am fi scris //node()[“Calculator”] am fi obținut TOATE nodurile, deoarece expresia din paranteze trebuie să fie true sau false, iar prezența unui string oarecare e echivalată cu true!
- Am fi mai aproape de rezultat cu o construcție de genul //node()[text()="Calculator"], ce dă toate nodurile care conțin un nod text egal cu “Calculator” (în timp ce punctul ne permite să returnăm chiar nodurile text ce sunt egale cu “Calculator”)

În plus mai sunt utile căile care traversează arborele pe orizontală:

/comanda/produs[3]/following::node()
...returnează toate nodurile care urmează în document după al 3-lea PRODUS din COMANDA (4, nu doar frații!)

/comanda/produs[3]/following-sibling::node()
...același lucru, dar se returnează numai frații ce urmează (3 soluții)

La fel avem și axa nodurilor precedente:

/comanda/produs[3]/preceding::text()
...returnează toate nodurile text ce preced al 3-lea PRODUS din COMANDA (5, din care 4 noduri invizibile)

/comanda/produs[3]/preceding::*/@*
...returnează toate atributele nodurilor precedente PRODUSULUI al 3-lea din COMANDA (5 soluții)

/comanda/produs**[@id=”p2”]**/preceding::node()**[preceding-sibling::comment()]**/..
..returnează părintele acelui element care e precedat de un frate de tip comentariu și care precede un PRODUS cu ID=”p2”, din COMANDA
...altfel spus:

- se coboară în COMANDA
- se caută elementul PRODUS cu atribut ID=”p2”, cu produs[@id=”p2”]
- se caută predecesorii săi de orice tip (2 la număr, un PRODUS și un comentariu), cu **preceding::node()**
- dintre aceștia se alege doar acel predecesor care are un frate precedent de tip comentariu, cu **[preceding-sibling::comment()]**
- pentru acesta se caută părintele, cu **..** (e chiar rădăcina)

Acesta e un exemplu complex care demonstrează detalii pe care începătorii le ignoră:

- Putem varia axa la fiecare pas al căii;
- Putem folosi axe în restricțiile dintre [];
- Putem folosi restricții la oricare pas al căii, nu doar la selecția rezultatului de pe ultimul pas.

Dacă o cale Xpath devine prea lungă, Oxygen vă deschide un panou suplimentar pentru continuarea tastării.

Căi cu restricții bazate pe calcule/funcții

Funcția **normalize-space** elimină caracterele albe (spațiu, Enter, Tab) de la începutul și sfârșitul unui string, iar în interiorul stringului păstrează maxim un spațiu între cuvinte.

//text()[normalize-space(.)=""]
...returnează toate nodurile invizibile din document (detectate prin faptul că, după ce li se aplică eliminarea de caractere albe, nu mai rămâne nimic din ele; 9 soluții)

//text()[normalize-space(.)!=""]
...returnează toate nodurile text care nu sunt invizibile (6 soluții)

//text()[contains(.,"3")]
...returnează toate nodurile text care conțin caracterul 3 (2 soluții, observați din nou punctul folosit pentru a pune o condiție asupra nodului curent)

//node()[contains(.,"3")]
...returnează toate nodurile indiferent de tip care conțin caracterul "3" (6 soluții! Pe lângă Idul și Pretul returnate și în cazul precedent, aici apar și toate nodurile care conțin acel ID și acel PRET, deci inclusiv PRODUSUL al 3-lea și elementul rădăcină: *de reținut – contains caută în toți descendenții, nu doar în conținutul textual direct*)

//node()[count(*)=3]
...returnează toate nodurile care au exact 3 elemente fiu (1 soluție)

//node()[count(node())=1]
...returnează toate nodurile care au exact 1 nod fiu (5 soluții)

Mai multe funcții se prezintă în secțiunea următoare (toate pot fi folosite atât ca restricții cât și pentru un calcul aplicat rezultatului final).

Căi cu rezultat bazat pe calcule/funcții

În această categorie intră căile asupra căror rezultate se aplică o funcție, de obicei una de agregare (sum, count etc.).

count(/comanda/node())
... returnează numărul de noduri fiu ale rădăcinii (*numărul 11, nu 11 soluții!*)

contains(/comanda/produs[3],"3")
... returnează true, deoarece al 3lea PRODUS din COMANDA conține caracterul 3

sum(//produs/@pret)
... returnează 300, suma atributelor PRET găsite în PRODUSE

sum(//@pret | //pret)
...returnează 600, suma tuturor atributelor și elementelor PRET

concat(//produs[2]/@id,//produs[2])
...returnează "p2Calculator", concatenarea rezultatelor celor două căi (Idul plus conținutul celui de-al doilea PRODUS)

name(//*[@id='p2'])
…returnează "produs", numele elementului care are atributul id egal cu 'p2'

name(/comanda/produs[1]/@*[2])
…returnează "id", numele atributului al doilea al primului PRODUS din COMANDA

Important: funcția name e mecanismul prin care putem afla numele unui element sau atribut la care am ajuns pe alte căi decât cu ajutorul numelui (poziție, relație cu alte noduri etc.)

string(/comanda)
... convertește rădăcina în string (rezultatul fiind toate conținuturile textuale concatenate între ele)

substring(//produs[@pret=200],2,3)
...returnează "alc", subșirul decupat de la poziția 2, de lungime 3, din valoarea PRODUSULUI cu PRET=200

substring-before(/comanda/text()[normalize-space(.)!=""], "com")
...returnează "Aceasta este o ", adică subșirul decupat de la început până la apariția lui "com", din acel nod text care nu este invizibil (există și varianta substring-after)

string-length(/comanda/produs[2])
...returnează 10, numărul de caractere din al doilea PRODUS

translate(//prettotal,"0","9")
...returnează valoarea lui PRETTOAL, în care substituie toate zerourile cu 9

translate(//produs[count(node())=1],"acr","xy")
...returnează "Cxlyulxto", obținut prin substituirea lui a cu x, a lui c u y și a lui r cu nimic (datorită faptului că al 3lea argument e mai scurt ca al doilea); modificarea se aplică aspra valorii acelui PRODUS care are exact 1 nod fiu (deci al 2lea)

boolean(/comanda/produs)
...există vreun PRODUS în COMANDA? (true, funcția boolean e folosită pentru a converti orice tip de interogare într-un booleană, dacă ne interesează doar existența unei soluții, nu și soluția în sine)

boolean(/comanda/*[position()=1 and self::produs])
...este PRODUS numele primului element fiu din COMANDA? (true, funcția boolean se combină cu o declarație de tip self ce exprimă numele nodului curent, pentru a-i testa numele)

Important: funcția position nu poate fi folosită pentru a calcula poziția unui nod!
Deci nu putem calcula position(/comanda/produs[id]) pentru a obține poziția între frați a PRODUSULUI ce are un fiu ID!

Vom apela la un artificiu, obținând poziția prin numărarea fraților precedenți:
count(/comanda/produs[id]/preceding-sibling::node())
...returnează 7 (numără toți frații precedenți, indiferent de tip)

Căi relative

Căile relative se calculează relativ la un nod curent. În Oxygen putem seta nodul curent printr-un click pe un marcator (care primește subliniere gri). Dați un click pe al treilea PRODUS.

Căile relative nu mai încep cu slash:

following::*
...returnează elementul următor (PRETTOTAL)

*
...returnează toate elementele fii ai nodului curent (3 soluții)

../*[1]
...returnează primul element fiu al părintelui nodului curent (primul PRODUS)

../node()[1]
...returnează primul nod fiu al părintelui nodului curent (primul nod invizibil)

count(*)
...returnează numărul de elemente fii ai nodului curent (3)

string-length(*[last()])
...returnează lungimea conținutului ultimului element fiu al nodului curent (4)

preceding-sibling::node()
...returnează precedenții frați ai nodului curent (7 soluții, inclusiv noduri invizibile și comentariul)

count(preceding-sibling::*[not(@*)])
...câți din frații precedenți nu au atribute (0)

preceding-sibling::*/@pret=100
...există între frații precedenți vreunul cu atributul PRET=100? (true)

id="p3"
...are nodul curent un fiu id cu conținutul "p3"? (true)

boolean(self::produs)
...are nodul curent numele produs? (true)

count(preceding-sibling::node())
...care e poziția, între frați, a nodului curent? (7) – deoarece nu putem folosi funcția position(), se apelează la acest truc de numărare a fraților precedenți

1.2 Validarea cu vocabulare XML Schema

XML Schema este cel mai popular limbaj pentru crearea de vocabulare XML. Un **vocabular XML** este un set de reguli ce stabilesc următoarele aspecte:

- ce marcatori se pot folosi într-un document XML;
- ce atribute pot să aibă acei marcatori;
- ce tipuri de date pot avea valorile atributelor şi conţinuturile textuale ale marcatorilor;
- cum se pot îmbina marcatorii între ei.

Regulile unui vocabular sunt suplimentare **regulilor de bună formare**, care sunt strict sintactice (ex.: toate atributele să aibă valori între ghilimele, să existe un element-rădăcină unic etc.). Regulile unui vocabular nu se referă doar la respectarea unor principii sintactice, ci stabilesc "dialectul", "limbajul" în care se poate scrie un document XML.

Practic, **regulile unui vocabular formează un LIMBAJ DE TIP XML**: De exemplu (X)HTML este un limbaj de tip XML, la baza căruia stau reguli precum:

- IMG să aibă ca atribut obligatoriu SRC şi ca atribute opţionale ALT, HEIGHT etc.
- Valorile atributelor HEIGHT, WIDTH să fie numerice
- TD să apară obligatoriu în interiorul lui TR, şi acesta în interiorul lui TABLE
- HTML să conţină HEAD şi BODY, exact în această ordine

Existenţa acestor reguli face posibil ca browserele să se pună de acord asupra modului în care interpretează marcatorii HTML.

Există o multitudine de astfel de vocabulare, fiecare cu regulile sale. Spre exemplu tipul de fişier .DOCX din Word este tot un limbaj de tip XML, numit WordML. Precum HTML, acesta conţine o serie de marcatori prin care se descriu structura şi formatările prezente într-un fişier Word. La fel există vocabulare pentru fişiere Excel, Powerpoint etc. Există şi vocabulare pentru fişiere grafice (SVG), pentru formule matematice (MathML), pentru descrierea de sunet (MusicXML) etc.

Un document ce respectă regulile impuse de un vocabular este considerat **valid în raport cu acel vocabular** (nu există validitate absolută, ci doar relativ la vocabularul ales). De exemplu: un document ce respectă regulile HTML, va fi valid faţă de regulile HTML (dar nu şi faţă de altele). În concluzie:

- regulile de bună formare sunt absolute (orice document XML trebuie să le respecte, indiferent de marcatori/attribute)
- regulile unui vocabular sunt relative (ele trebuie respectate doar de documentele ce vor fi folosite de utilizatorii acelui vocabular).

După ce un vocabular se creează, prin definirea regulilor sale, acele reguli se fac publice şi se promovează pentru a permite oricui să creeze programe care pot procesa documente valide în raport cu vocabularul respective. De exemplu: browserele nu sunt altceva decât programe capabile să proceseze documente scrise conform cu vocabularul HTML; Word este un program capabil să deschidă documente scrise conform cu vocabularul WordML.

Cu alte cuvinte, ciclul de viaţă al vocabularelor este următorul:

- Un vocabular se creează atunci când cineva defineşte **un set de reguli pentru vocabular**. Acestea se pot crea cu DTD, XML Schema, Schematron şi alte limbaje similare. În această carte, vom folosi XML Schema în Oxygen pentru a crea un vocabular;
- Apoi vocabularul este publicat on-line şi, dacă restul lumii îl consideră util, va crea **documente valide în raport cu acel vocabular**. Toată lumea care adoptă vocabularul respectiv va şti la ce să se aştepte când primeşte un astfel de document. În această carte, vom crea unele documente pentru vocabularul nostru şi le vom testa validitatea în raport cu regulile sale;
- Dacă vocabularul devine suficient de popular, cei care îl adoptă vor începe să creeze **programe capabile să prelucreze documente de acel tip**. De exemplu, browserele au apărut din necesitatea de a afişa într-un mod prietenos conţinutul documentelor HTML. MS Word există din necesitatea de a permite utilizatorilor să lucreze cu fişiere WordML (.docx).
    - În cele mai multe cazuri, utilizatorii nici nu ajung în contact cu marcatorii XML! Utilizatorii MS Office rareori intră în contact cu codul WordML. Utilizatorii unui browser rareori sunt interesaţi să vadă codul sursă al unei pagini HTML.
    - Uneori vocabularul e creat de aceiaşi autori care creează şi programele cu care se procesează documentele vocabularului. De exemplu Microsoft a creat atât vocabularul WordML (.DOCX) cât şi programul (Word) care poate deschide fişiere de acel tip. Utilizatorii Word sunt cei care creează documentele!
    - Alteori, creatorii vocabularului, ai documentelor şi ai programelor sunt diferiţi! Pentru HTML, vocabularul a fost creat de Consorţiul Web (W3C), documentele (paginile Web) sunt create de designerii Web, iar programele ce pot interpreta acele documente (browserele) sunt create de producătorii de browsere.
    - Alteori, creatorii vocabularului, ai documentelor şi ai programelor sunt aceiaşi! Adesea într-un parteneriat ce include mai multe companii, acestea se pun de acord asupra unui vocabular, apoi creează programe interne capabile să prelucreze documente de acel tip, apoi încep să schimbe între ele documente (de ex. rapoarte, facturi) conforme cu acel vocabular.
- Programele ce vor procesa documente XML se creează cu ajutorul funcţiilor specializate pentru extragerea de informaţii din XML.

În general se spune că XML este un limbaj flexibil pentru că ne permite să folosim ce marcatori dorim şi ce atribute dorim. Realitatea este însă alta: XML nu e un limbaj, ci un set de reguli (de bună formare) ce ne permit:

(a) să creăm propriul limbaj (vocabular) cu ce marcatori/atribute dorim;
(b) apoi să creăm documentele ce respectă acel vocabular (folosind doar marcatorii/atributele definite la primul pas);
(c) apoi să creăm programe ce pot prelucra documente precum cele create la pasul anterior.

Cel mai popular limbaj pentru crearea de vocabulare este XML Schema. Oxygen ne permite să utilităm acest limbaj în mod vizual. Vom folosi în continuare Oxygen pentru a crea un vocabular minimal, apoi documente valide conform cu acesta. Regulile sunt următoarele:

Elementul rădăcină să fie **Comanda:**

- **Comanda** să conțină obligatoriu maxim 3 elemente **Produs** urmate de 1 element opțional **Onorata:**
    - Fiecare **Produs** să aibă 2 atribute obligatorii **(CodProdus, Pret)** și un nod text obligatoriu **(denumirea produsului);**
        - **CodProdus** să aibă valoare string unică **("cheie primară")** de exact 5 caractere;
        - **Pret** să aibă valoare numerică pozitivă întreagă**;**
        - **Nodul text** ce exprimă denumirea să accepte doar una din valorile **Televizor, Calculator, Ipod;**
    - **Onorata** să conțină un nod text obligatoriu, cu o valoare booleană, indicând dacă s-a onorat comanda sau nu.

XML Schema se bazează pe noțiunea de TIP, ceva mai largă decât la alte limbaje. TIPURILE pot fi:

- TIP PREDEFINIT: aici intră orice tip de date clasic (string, integer, etc.) plus unele propuse de XML Schema (ID = stringuri unice, Name = stringuri fără spații, token = stringuri cu maxim un spațiu între cuvinte, etc.);
- TIP SIMPLU: aici intră orice subset al unui TIP PREDEFINIT (de ex. un interval de valori integer, o listă de stringuri prestabilite etc.) sau o combinaţie (reuniune) între astfel de subseturi;
- TIP COMPLEX: un astfel de tip este o **combinaţie de marcatori** împreună cu atributele şi conţinuturile lor, **ce se dorește a fi reutilizată** (deci o structură XML reutilizabilă).

Crearea unui vocabular are fazele:

- Se creează TIPURI, în care se stochează regulile vocabularului
- Se definesc marcatori şi atribute corespunzătoare acelor TIPURI. În general TIPURILE SIMPLE şi PREDEFINITE se leagă de atribute şi/sau noduri text, iar cele COMPLEXE se leagă de marcatori.

În felul acesta, **regulile unui vocabular devin modulare** – regulile se pot edita separat de marcatori.

*Observație: În general se evită crearea unui TIP pentru elementul rădăcină, având în vedere unicitatea lui,nu se prea pune problema reutilizării sale. Sunt totuși situații în care și structura rădăcinii se poate reutiliza ducând la **recursivitate**, când copiii rădăcinii moștenesc structura rădăcinii, nepoții la fel, etc. ramificând arborele DOM în maniera fractalilor, teoretic la infinit. Practic nu se merge la infinit dacă structurii recursive i se alocă un caracter opțional (astfel încât nodurile de pe ultimul nivel să poată să nu mai conțină nimic).*

Faza 1. Planificarea tipurilor

Tipurile se definesc în manieră bottom-up, pornind de la nodurile simple ale viitoarelor documente (atributele și nodurile text). În cazul de față avem nevoie de:

1. Tipuri pentru atribute:

- **CodProdus:** avem nevoie de un TIP SIMPLU, obţinut prin: restricţia "maxim 5 caractere" aplicată TIPULUI PREDEFINIT ID (valoare string unică);

- **Pret**: avem nevoie de TIPUL PREDEFINIT "numere pozitive întregi"

2. Tipuri pentru nodurile simple (care conțin doar text, fără atribute):
- Pentru denumirile de produse: avem nevoie de un TIP SIMPLU, obţinut prin: enumerarea a 3 valori prestabilite (Televizor, Calculator, Ipod), fiecare având TIPUL PREDEFINIT string;
- **Onorat**: avem nevoie de TIPUL PREDEFINIT boolean.

3. Tipuri pt noduri complexe (care conțin și altceva decât text – atribute sau elemente copil):
- **Produs**: avem nevoie de o structură alcătuită din atribute (CodProdus, Pret) și denumirea produsului ca nod text. Pentru toate 3 vom avea deja tipuri definite de la punctele precedente)
- **Comanda**: chiar dacă e un element complex, faptul că e rădăcină atrage după sine faptul că va fi unic! Deci nu are sens reutilizarea structurii sale, deci nu are sens crearea unui TIP (există totuşi situaţii în care TIPUL rădăcinii e reutilizat: atunci când dorim să permitem documente recursive, unde rădăcina poate conţine elemente similare ei însăşi; nu e cazul aici)

Ca regulă generală:
- TIPURILE SIMPLE se creează pentru atribute sau elemente ce vor conține doar text;
- TIPURILE COMPLEXE se creează pentru elemente care pot avea și altceva în afară de conținut textual (atribute sau elemente copil).

Faza 2. Crearea vocabularului

Creăm în Oxygen un fișier de tip XML Schema. Spre deosebire de documentele XML, vocabularele în Oxygen pot fi create foarte comod în mod vizual (modul Design) fără a cunoaşte neapărat limbajul XML Schema!

În modul Design, cea mai mare parte din muncă se realizează în meniul click dreapta, cu primele două opțiuni:
- <u>New Global</u> ne permite să construim TIPURILE;
- <u>Edit Attributes</u> ne permite să configurăm TIPURILE sau alte componente ale vocabularului

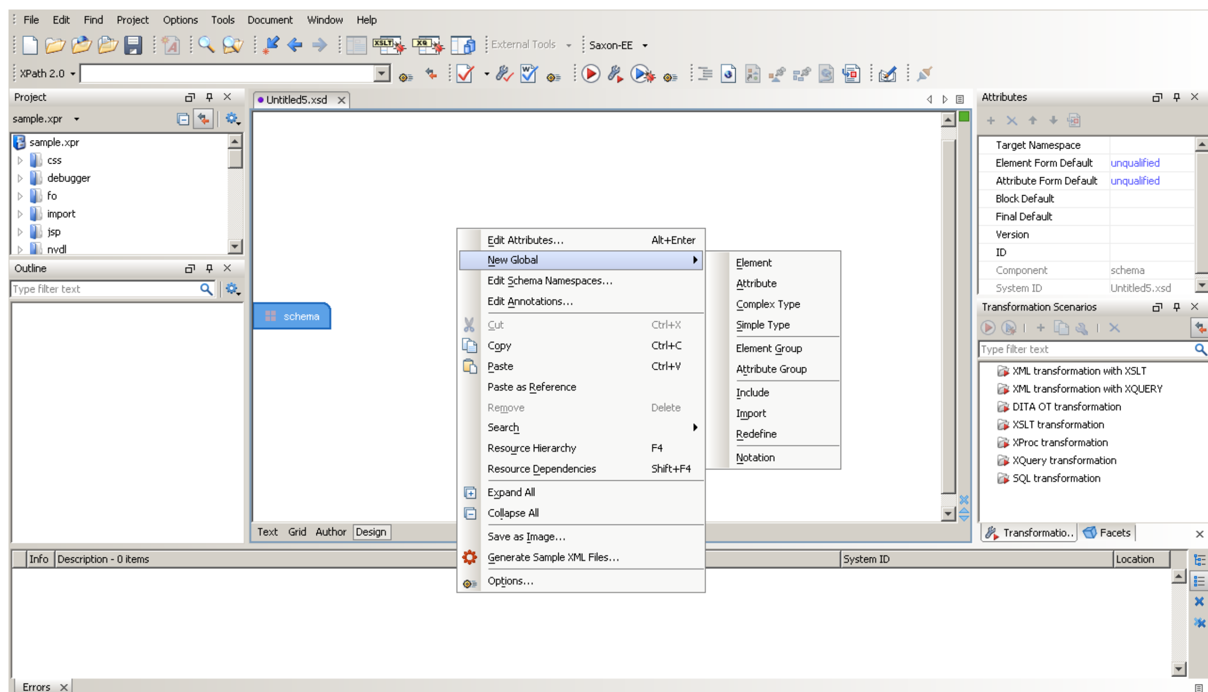


Figura 7 Interfața grafică Oxygen (Design) pentru crearea vocabularelor XML Schema

Avertisment:

- Edit Attributes NU se referă la **atributele XML**, ci la diverse proprietăți configurabile în Oxygen (e o coincidență de denumire care riscă să vă ducă în eroare);
- Panoul Attributes din dreapta oferă de obicei aceleași opțiuni cu Edit Attributes.

Faza 3. Crearea tipurilor

Se începe cu TIPURILE SIMPLE, într-o abordare bottom-up, de la simplu la complex:

TIPUL SIMPLU pentru denumirile de produse:

- click dreapta – New Global – Simple Type
- dăm tipului un nume (ModelDenumiri)

Trebuie să precizăm din ce TIP PREDEFINIT îl vom crea, și ce filtru vom aplica:

- click dreapta pe ModelDenumiri – Edit Attributes
- în jumătatea de sus a ferestrei alegem TIPUL PREDEFINIT: Base Type = xs:string
- în secțiunea Facets definim filtrul: Enumerations, apoi cu click dreapta – Add adăugăm pe rând valorile permise

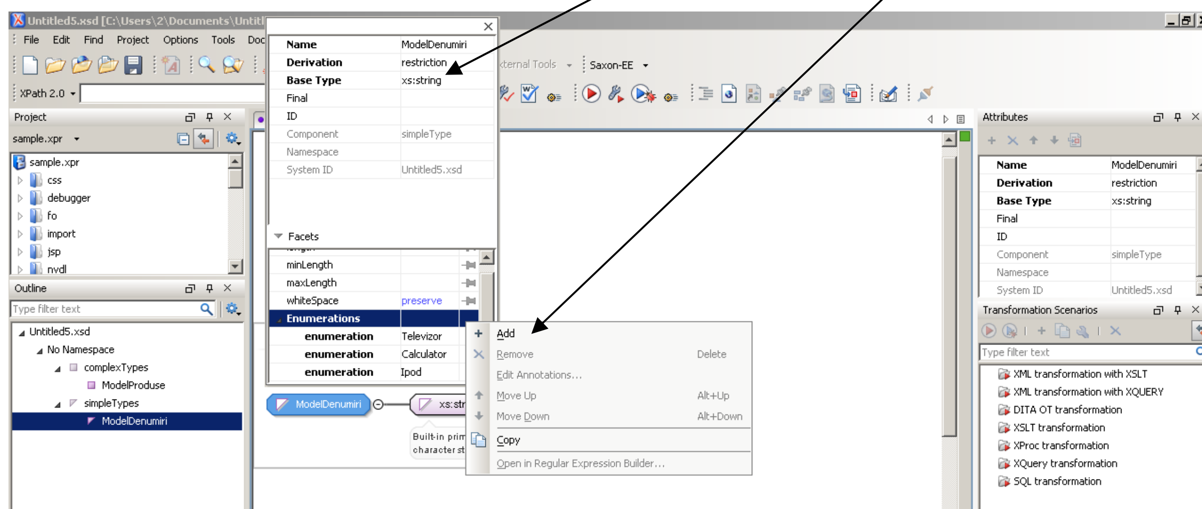


Figura 8 Crearea unui tip simplu XML Schema prin interfața grafică Oxygen XML

TIPUL SIMPLU pentru codurile de produse:

- click dreapta (pe grupul SCHEMA) – New Global – Simple Type
- dăm tipului numele ModelCoduri
- click dreapta pe ModelCoduri – Edit Attributes
- TIPUL PREDEFINIT: BaseType=xs:ID
- filtru: length 5

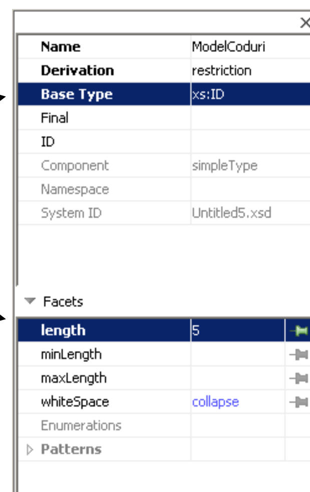


Figura 9 Configurarea tipului ModelCoduri

TIPUL COMPLEX pentru Produs:

- click dreapta pe SCHEMA – New Global – Complex Type;
- dăm tipului un nume (ModelProduse)

Mai întâi ne ocupăm de conținutul textual al viitoarelor Produse:

- click dreapta pe ModelProduse – Edit Attributes;
- alegem TIPUL conținutului: BaseType = ModelDenumiri (iată cum integrăm TIPURILE SIMPLE într-o structură de TIP COMPLEX); precizarea acestui tip va impune și obligativitatea denumirii (fiind definită clar lista de valori, nu e permis șirul vid, deci absența conținutului!)

Apoi ne ocupăm de atribute:

- click dreapta pe ModelProduse – Append Child – Attribute
- dăm atributului un nume (CodProdus), apoi ne ocupăm de el:
    - o alegem TIPUL atributului: <u>Type=ModelCoduri</u> (de unde va moșteni toate restricțiile)
    - o alegem obligativitatea: <u>Use = required</u>
- click dreapta pe ModelProduse – Append Child – Attribute
- dăm atributului nume: Pret
    - o îi alegem TIPUL PREDEFINIT: Type=xs:positiveInteger
    - o alegem obligativitatea: Use=required

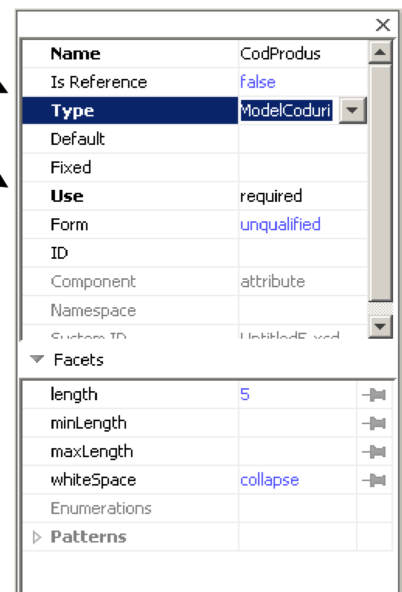


Figura 10 Stabilirea restricțiilor pe atributul CodProdus

În acest moment toate TIPURILE sunt create:

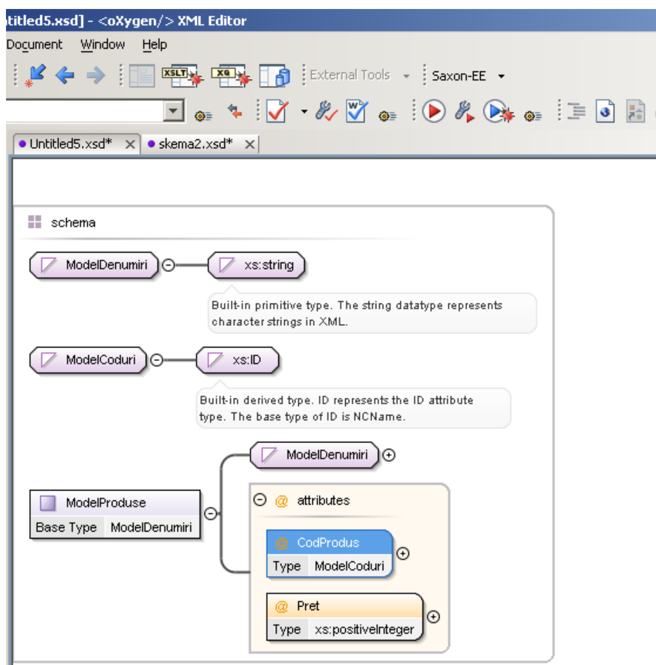


Figura 11 Tipurile simple și complexe XML Schema create în Oxygen XML

Faza 4. Crearea rădăcinii

- Click dreapta pe SCHEMA – New Global – Element
- Îi dăm nume: Comanda

Structura internă va fi o **SECVENȚĂ = șir de elemente într-o anumită ordine**

- click dreapta pe Comanda – Append Child – Sequence
- click dreapta pe SECVENȚĂ – Append Child – Element
    - o îi dăm nume: Produs;
- click dreapta pe SECVENȚĂ – Append Child – Element

   - îi dăm nume: Onorata

Le stabilim limitele de ocurență, caracterul opțional (min.ocur.=0) sau obligatoriu (min.ocur.=1), precum și TIPUL conținutului:

- click dreapta pe Produs – Edit Attributes:
   - TIP conținut: Type=ModelProduse
   - Min.Occur. =1, Max.Occur.=3 (obligatoriu, maxim de 3 ori)
- click dreapta pe Onorata – Edit Attributes:
   - TIP: Type=xs:boolean
   - Min.Occur.=0, Max.Occur.=1 (opțional, maxim o dată)

Vocabularul final arată astfel:

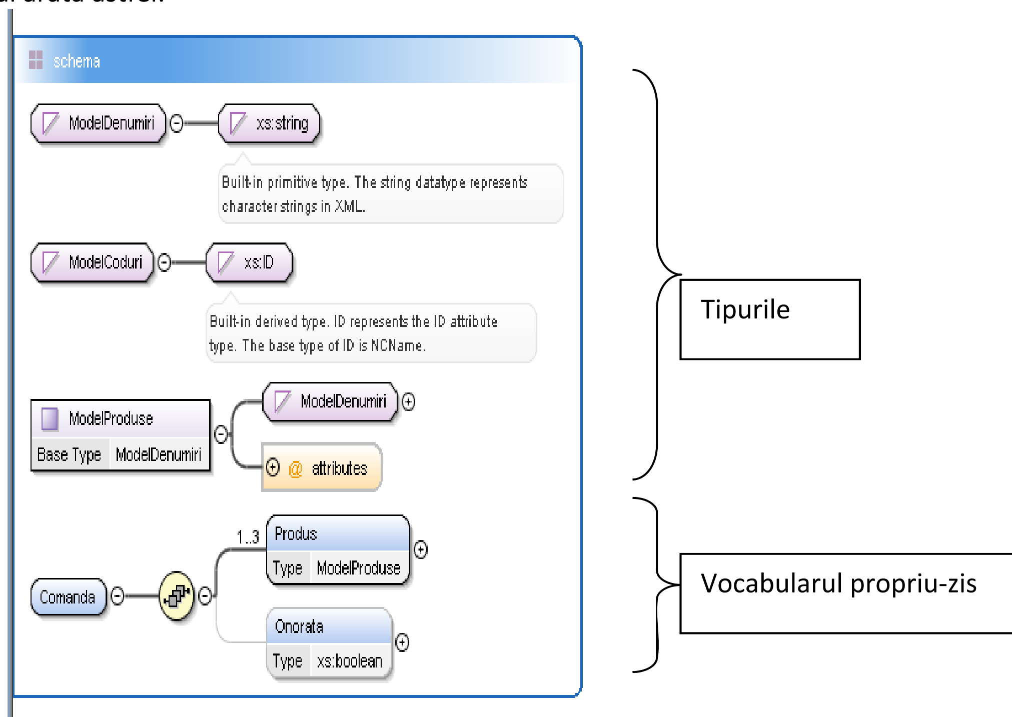


Figura 12 Vocabularul final XMLSchema creat în Oxygen XML

În modul Text puteți vizualiza codul sursă. Se poate observa pe codul sursă că limbajul XML Schema este el însuși un vocabular XML!

Salvăm vocabularul cu numele vocabular.xsd

Faza 5. Validarea

Creăm un document XML nou. De data aceasta, la creare apăsăm Customize pentru a indica vocabularul, în caseta Schema URL.

Încă de la creare, documentul va conține un element Produs, iar rădăcina va fi însoțită de o serie de atribute ce indică unde s-a salvat vocabularul (**noNamespaceSchemaLocation**, calea fişierului va fi diferită pe calculatoarele voastre şi trebuie păstrată şi în exemplele următoare!):

```
<?xml version="1.0" encoding="UTF-8"?>
<Comanda xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="file:/C:/Users/2/Documents/vocabular.xsd">
   <Produs CodProdus="" Pret=""></Produs>
</Comanda>
```

Prezența unui Produs vid e explicabilă prin caracterul său obligatoriu. Când un document e construit pe baza unui vocabular, se pot genera automat o serie de noduri obligatorii sau valori implicite.

La apăsarea butonului de validare primim numeroase erori:

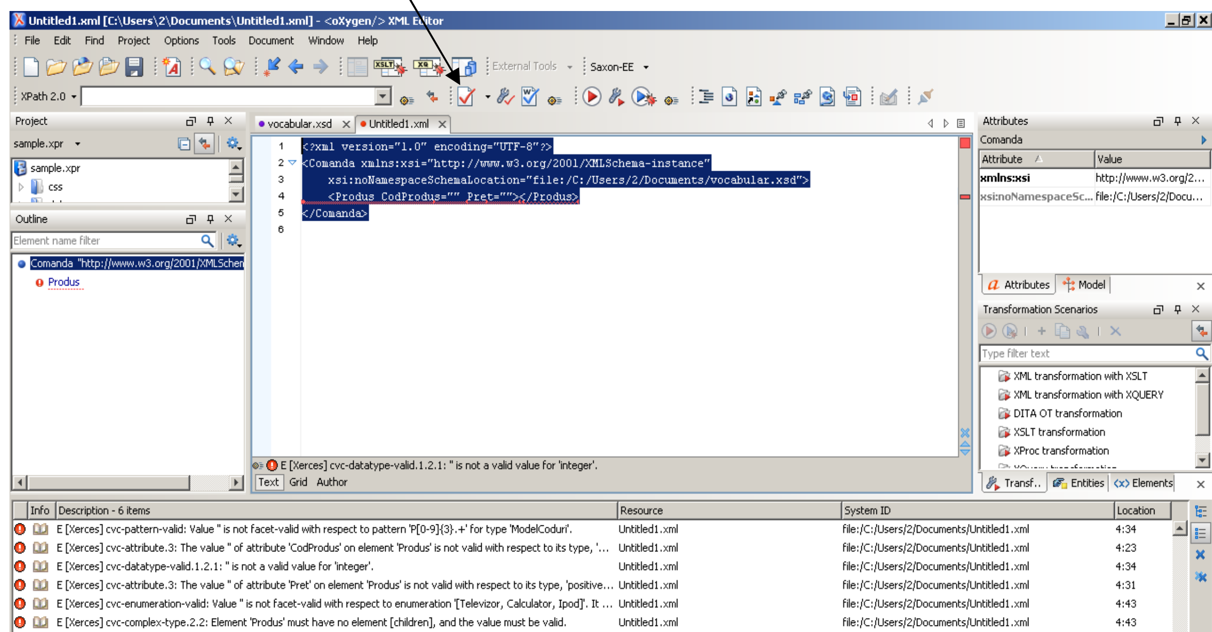


Figura 13 Validarea unui document XML în raport cu vocabularul creat

Putem testa pe rând regulile vocabularului, scriind un document corect, apoi încălcând regulile una câte una, pentru a ne convinge de efectul acestora.

Următorul este un document valid:

```
<?xml version="1.0" encoding="UTF-8"?>
<Comanda xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file:/C:/Users/......aici sa aveti calea vocabularului ">
  <Produs CodProdus="aaaaa" Pret="2">Televizor</Produs>
  <Produs CodProdus="bbbbb" Pret="4">Ipod</Produs>
  <Produs CodProdus="ccccc" Pret="10">Ipod</Produs>
</Comanda>
```

Deși lipsește elementul Onorata, nu e o problemă, acesta fiind definit cu caracter opțional. Nici faptul că valoarea Ipod se repetă nu e o problemă, important e să nu se iasă din lista de valori permise. Tot valid e și următorul document:

```
<?xml version="1.0" encoding="UTF-8"?>
<Comanda xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file:/C:/Users/......aici sa aveti calea vocabularului ">
  <Produs CodProdus="aaaaa" Pret="2">Televizor</Produs>
  <Onorata>true</Onorata>
</Comanda>
```

Faza 6. Modificarea vocabularelor

Nu e obligatoriu ca la crearea unui vocabular să se lucreze cu TIPURI, așa cum am lucrat în acest mod. Se pot impune toate restricțiile direct pe nodurile vocabularului, fără a mai crea tipuri.

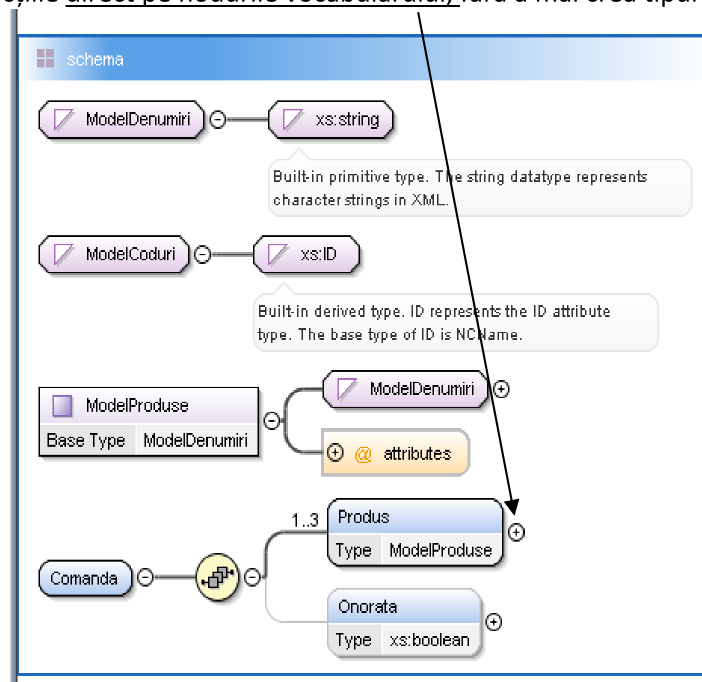


Figura 14 Adăugarea de restricții direct pe nodurile vocabularului

Totuși, tipurile oferă avantaje legate de reutilizare și modularitate:

- mai multe atribute/elemente pot să aibă același tip, deci aceeași gamă de valori/structură internă permisă (vezi exemplul următor);
- se pot crea tipuri din alte tipuri (s-a văzut deja cum ModelProduse include ModelDenumiri și ModelCoduri);
- se pot aduce modificări la un tip fără a afecta restul vocabularului (vezi mai jos).

De exemplu, în cazul de față dorim următoarele modificări:

- **Codurile de produse să aibă structura: litera P, urmată de exact 3 cifre, urmate de cel puțin încă un caracter oarecare.**
- **În loc de Onorata să poată să mai apară un grup de maxim 3 elemente ProdusIndisponibil, cu aceeași structură internă precum elementele Produs.**

Pentru a modifica structura codurilor de produse e suficient să modificăm TIPUL ModelCoduri – modificarea se va propaga peste tot în vocabular unde avem atribute (sau elemente!) de acest tip.

- Click dreapta pe ModelCoduri – Edit Attributes
- Renunțăm la restricția length=5 (din panoul Facets)
- În locul ei folosim restricția **pattern = P[0-9]{3}.+**

Iată o calitate foarte importantă a XML Schema – posibilitatea de a constrânge valori cu expresii regulate!

Expresia regulată exprimă exact structura pe care o doream: să înceapă cu P, să urmeze un caracter din intervalul [0-9] repetat de exact 3 ori, apoi un caracter oarecare repetat minim o dată (punctul e locțiitor pt un caracter, + e indicator de repetiție cu obligativitate: minim o dată).

Pentru a preciza că elementul Onorata poate fi ALTERNAT cu un grup de elemente ProdusIndisponibil, înlocuim Onorata cu un nod de tip CHOICE:

- Click dreapta pe noul de tip Secvenţă din Comanda – Append Child – Choice
- De nodul Choice se vor lipi toate alternativele între care oferim posibilitatea de a alege:
    - cum Onorata va fi una din variante, o tragem cu mouseul până se lipește de nodul Choice
    - click dreapta pe nodul Choice – Append Child – Element și denumim noul element cu numele ProdusIndisponibil.

Mai trebuie doar să precizăm că ProdusIndisponibil reutilizează aceeași structură precum elementele Produs, apoi să-i stabilim ocurențele:

- click dreapta pe ProdusIndisponibil – Edit Attributes
    - Min.Occurs=1, Max. Occurs=3
    - Type=ModelProduse

Forma finală a vocabularului va fi următoarea:

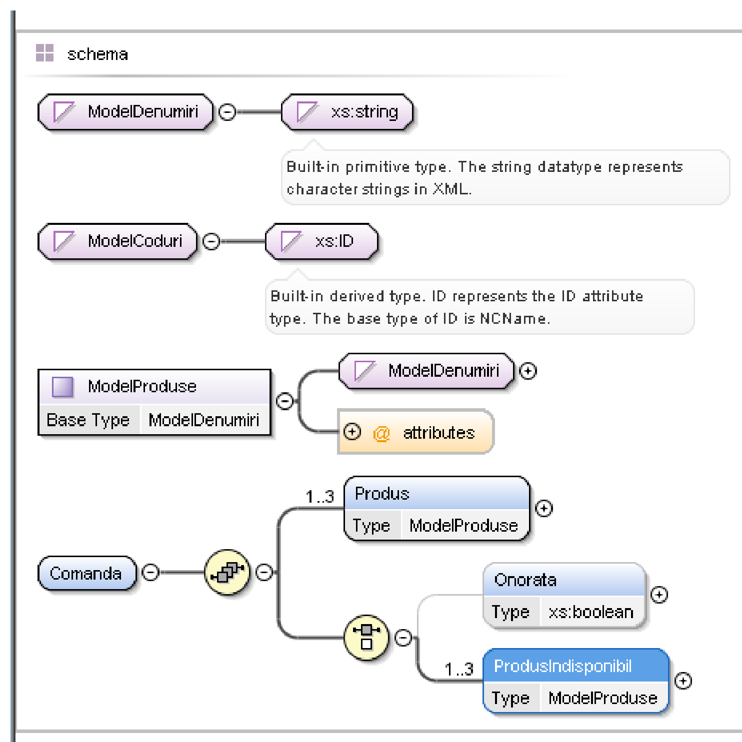


Figura 15 Forma finală a vocabularul modificat

Un exemplu de document valid ar fi următorul:

```
<?xml version="1.0" encoding="UTF-8"?>
<Comanda xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="file:/C:/Users/......aici sa aveti calea vocabularului">
   <Produs CodProdus="P999xx" Pret="200">Televizor</Produs>
   <Produs CodProdus="P1111" Pret="50">Ipod</Produs>
   <ProdusIndisponibil CodProdus="P123q" Pret="400">Calculator</ProdusIndisponibil>
</Comanda>
```

Dar documentul de mai jos?

```
<?xml version="1.0" encoding="UTF-8"?>
<Comanda xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=" file:/C:/Users/......aici sa aveti calea vocabularului ">
   <Produs CodProdus="P999xx" Pret="200">Televizor</Produs>
</Comanda>
```

Și acesta e valid, chiar dacă lipsesc atât ProdusIndisponibil cât și Onorata! Asta din cauză că Choice ne permite să alegem între 1-3 elemente ProdusIndisponibil și 0-1 elemente Onorata. Practic lipsa părții de după Choice înseamnă că am optat pentru Onorata dar am profitat de caracterul opțional și nu l-am mai pus deloc.

Faza 7. Validarea în PHP

Majoritatea limbajelor de programare, inclusiv PHP, oferă facilitatea de validare XML Schema, dacă au acces atât la vocabular (fişierul XML Schema) cât şi la documentul XML. Într-un scenariu realist, de obicei vocabularul este stocat undeva pe server, iar documentul XML soseşte de la o organizaţie parteneră. Înainte de a începe prelucrarea sa, un script PHP trebuie să verifice mai întâi dacă documentul sosit e conform cu regulile vocabularului. În continuare vom presupune că atât vocabularul cât şi documentul sosit au fost deja salvate pe serverul nostru.

Pentru un astfel de exemplu, aveţi de nevoie de sistemul client-server XAMPP[9] instalat, cu serverul Apache pornit şi fişierele de mai jos salvate în htdocs.

Pas1. În htdocs, salvaţi mai întâi vocabularul anterior creat în Oxygen (cu ultimele modificări), într-un fişier numit vocabular.xsd:

Pas2. În acelaşi folder, salvaţi următorul document XML, într-un fişier cu numele document.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<Comanda>
   <Produs CodProdus="P999xx" Pret="200">Televizor</Produs>
   <Produs CodProdus="P1111" Pret="50">Ipod</Produs>
   <ProdusIndisponibil CodProdus="P123q" Pret="400">Calculator</ProdusIndisponibil>
</Comanda>
```

Pas3. În acelaşi folder, creaţi scriptul care realizează validarea, salvat cu numele validare.php:

```
<?php
        $doc=new DOMDocument();
        $doc->load('document.xml');
        print $doc->schemaValidate('vocabular.xsd');
?>
```

La executarea scriptului prin localhost, veţi primi rezultatul 1, care semnifică validitatea documentului. Înlocuiţi codul primului produs cu "x999xx" şi veţi primi rezultatul 0, împreună cu erorile de validare:

- faptul că noul cod nu respectă expresia regulată (nu începe cu P)
- faptul că TIPUL ModelCoduri nu a fost respectat (din aceeaşi cauză, eroarea se propagă în toate tipurile care se bazau pe ea).

1.3 Transformări XSLT

XSLT este limbajul de transformare a documentelor XML și se bazează pe Xpath pentru a extrage informații dintr-un **document sursă** și a le pune într-un **document rezultat** (cu structură diferită).

- Sursa trebuie să fie un document XML bine format
- Rezultatul NU trebuie să fie XML bine format. XSLT e folosit adesea pentru a genera pagini HTML din date disponibile în format XML (deci poate fi folosit și ca mecanism AJAX, pentru a converti un răspuns XML de la server în cod HTML gata de afişat).

Pornim de la următorul document XML (salvat cu numele comanda.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<comanda>
```

[9] disponibil la adresa https://www.apachefriends.org/download.html

```
   <!-- acesta e un comentariu -->
   <produs denumire="Televizor" id="p1" pret="100"/>
   <produs id="p2" pret="200">Calculator</produs>
   <produs>
      <id>p3</id>
      <pret>300</pret>
      <denumire>Ipod</denumire>
   </produs>
   <prettotal>600</prettotal>
   Aceasta este o comanda de produse
</comanda>
```

Creăm în Oxygen următoarea foaie XSLT (New Document – XSLT stylesheet, salvată cu numele transformare.xsl):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/">Text oarecare</xsl:template>
</xsl:stylesheet>
```

O foaie XSLT conține una sau mai multe **reguli de substituire** cu două componente:

```
<xsl:template match="/">
Text oarecare
</xsl:template>
```

match este calea Xpath a elementelor care vor suferi substituția

conținutul lui **xsl:template** este noul conținut care va intra în locul celui detectat de **match**.

Exemplul de față selectează întreg documentul (calea Xpath "/" reprezintă documentul în ansamblul său) și îl înlocuiește cu stringul "Text oarecare". Observaţi că XSLT este la rândul său un LIMBAJ DE TIP XML (vocabular): instrucţiunile sale sunt conforme regulilor de bună formare şi aparţin unui vocabular de marcatori impus de standardul XSL. Vom vedea mai târziu că de fapt avem de a face cu un limbaj de programare.

Pentru a executa transformarea, Oxygen impune să se creeze (1) documentul sursă (de tip XML), (2) transformarea (un fişier XML ce conţine regulile de substituire XSLT) şi (3) un scenariu de transformare prin care se precizează care e transformarea şi pe ce sursă trebuie aplicată.

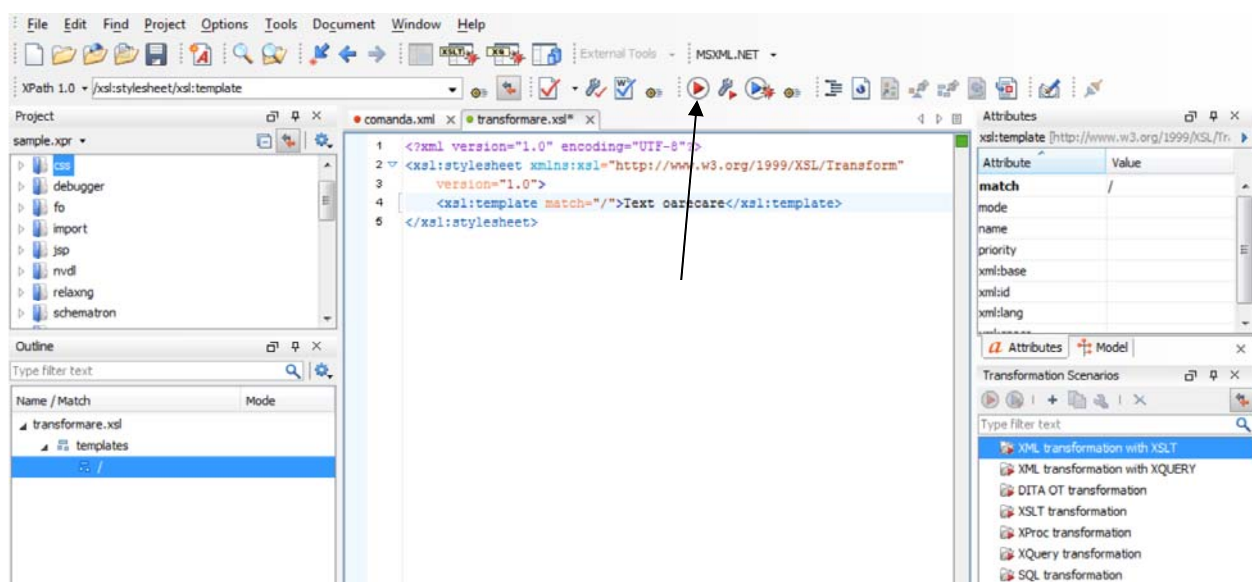


Figura 16 Execuția transformărilor XSLT în Oxygen

Crearea scenariului e solicitată când încercăm să executăm transformarea.

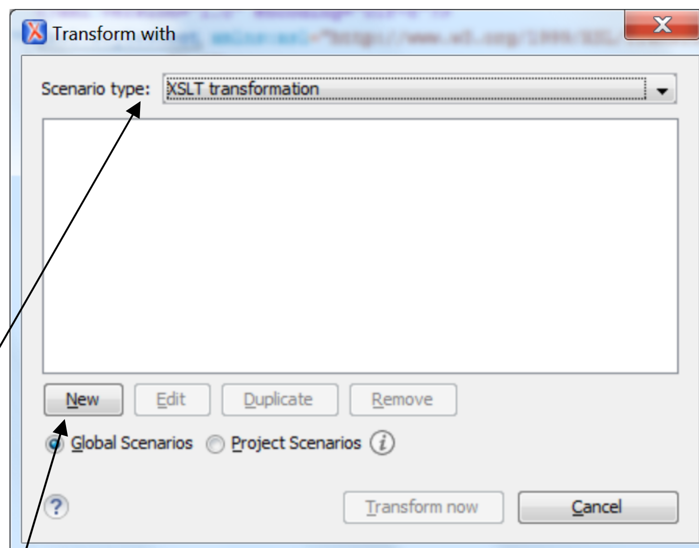


Figura 17 Crearea unui scenariu de transformare

Tipul scenariului va fi **XSLT transformation** dacă documentul activ din Oxygen e chiar transformarea (dacă e XMLul original, alegem tipul **XML transformation with XSLT**).
Creăm un scenariu nou.

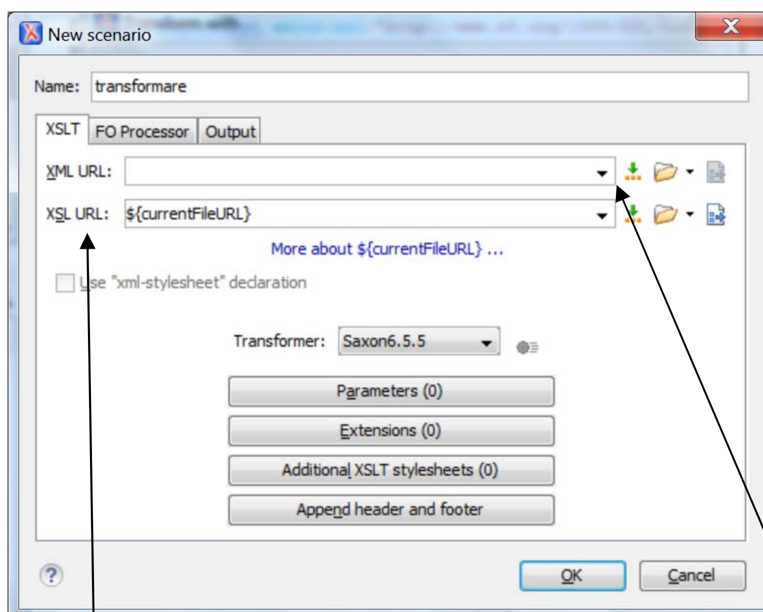


Figura 18 Configurarea intrărilor pentru scenariul de transformare

Setările esențiale sunt în prima și ultima categorie. *Categoria FO Processor se folosește doar dacă facem transformări de tip XSL-FO (o alternativă la CSS, dedicată formatării de cod XML).*

Avem deja completată calea foii XSLT curente (codul **currentFileURL** e suficient). Trebuie să selectăm manual adresa XMLului original.

În acest fel am precizat ce se va transforma (comanda.xml) și cum se va transforma (foaia curentă din Oxygen). Mai trebuie să precizăm ce să se întâmple cu rezultatul transformării, în categoria Output:

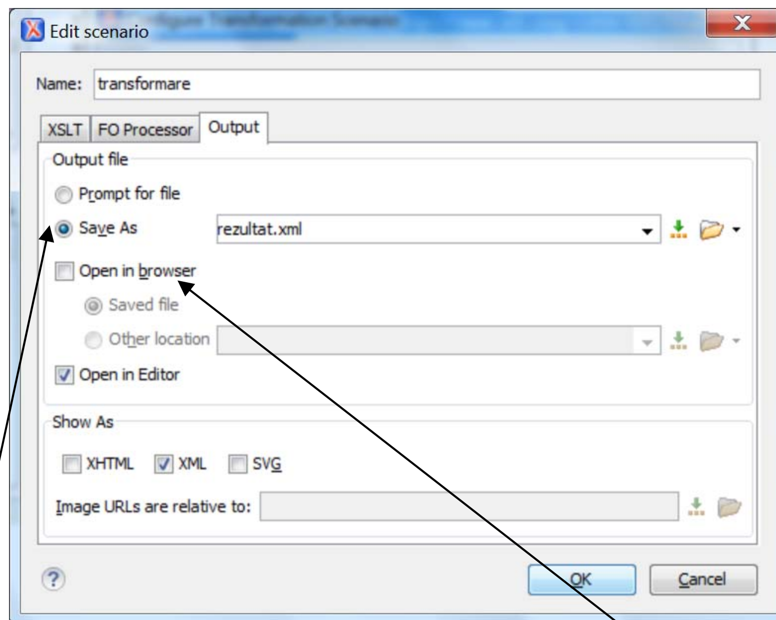


Figura 19 Configurarea ieșirilor pentru scenariul de transformare

Decidem să salvăm fișierul cu numele rezultat.xml și să deschidem imediat rezultatul în Oxygen. Apoi apăsăm OK și Transform now. Rezultatul va fi:

```
<?xml version="1.0" encoding="utf-8"?>Text oarecare
```

Se observă că:

- Nu e obligatoriu ca XSLT să returneze documente XML bine formate. În acest exemplu s-a încălcat regula care impune existenţa unui element rădăcină;
- Nu e obligatoriu să returneze informaţii din documentul original. În acest exemple conţinutul din sursă s-a substituit complet cu altceva. Totuşi de obicei rostul lui XSLT e să extragă anumite date de interes din sursă și să le reorganizeze într-o structură nouă (de exemplu o pagină HTML).

O concluzie importantă este că **XSLT nu conservă implicit conținutul fișierului original**! Dacă nu punem nimic în regula de substituire, se returnează un document gol. *Dacă vrem să conservăm întocmai unele elemente din fișierul original, trebuie să definim explicit o **regulă de conservare** (=o regulă de substituire în care elementele dorite din fişierul sursă sunt substituite cu ele însele).* Revenim mai târziu la aceste reguli de conservare.

Reguli de substituire iterative

Substituirile iterative sunt cele mai populare în XSLT, pentru că permit să se parcurgă (cu un ciclu FOR) toate rezultatele unei interogări Xpath și să se aplice pe fiecare din acestea o transformare. Adesea ele sunt îmbinate cu teste de tip IF/CASE care permit tratarea diferită a unor elemente în funcţie de o condiţie (exprimată ca o interogare Xpath booleană).

În exemplul următor, pentru fiecare rezultat al interogării comanda/produs (deci fiecare produs din comanda) se va returna cuvântul *Produs* scris cu italic (în HTML):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/">
            <xsl:for-each select="comanda/produs">
<i>Produs</i>
```

```
</xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

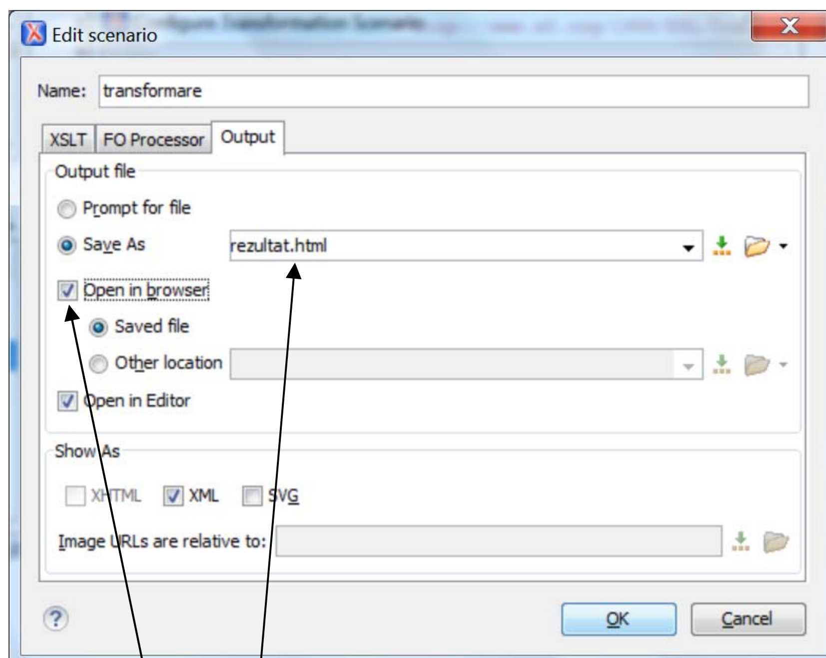


Figura 20 Modificarea ieșirilor la scenariul de configurare

La crearea scenariului de transformare aplicaţi următoarele configurări:

- de data aceasta generăm cod HTML;
- am bifat și opțiunea de afișare în browser a rezultatului salvat.

Rezultat:

```
<?xml version="1.0" encoding="utf-8"?><i>Produs</i><i>Produs</i><i>Produs</i>
```

Modificăm exemplul pentru a afișa cuvintele unul sub celălalt și pentru a concatena de fiecare dată o numerotare (*Produs0, Produs1* etc.), adică echivalentul unui ciclu FOR de genul următor (dacă am fi în PHP):

```
for ($i=0;$i<3;$i++)
         print "<i>Produs</i>".$i."<br/>";
```

Pentru asta vom avea nevoie de lucru cu variabile. O problemă spinoasă pentru începători e **modul în care XSLT tratează conceptul de variabilă**, diferit de alte limbaje:

Variabilelor li se poate atribui valoare O SINGURĂ DATĂ, deci ar putea fi considerate "constante". Totuşi, valoarea atribuită poate varia, atunci când provine dintr-o interogare Xpath cu mai multe rezultate.

Inițializarea unei variabile se realizează cu una din construcțiile:

```
<xsl:variable name="i" select="0"/>
```

dacă valoarea este una simplă (indiferent că e tastată direct sau rezultată dintr-o interogare Xpath)

```
<xsl:variable name="i">
<i>Produs</i>
</xsl:variable>
```

dacă valoarea este complexă (cod XML)

În funcție de complexitatea variabilei, şi accesarea sa se face în moduri diferite:

```
<xsl:value-of select="$i"/>
```

dacă valoarea e simplă, sau dacă e cod XML din care vrem să preluăm doar conținutul textual, ignorând marcatorii

```
<xsl:copy-of select="$i"/>
```

dacă valoarea e complexă (cod XML) și vrem să o preluăm integral, cu tot cu marcatori

Rostul uzual al variabilelor este să asigure **reutilizarea unui fragment de cod XML**. De exemplu precedenta transformare poate să aibă și următoarea formă:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:variable name="i">
            <i>Produs</i>
   </xsl:variable>
   <xsl:template match="/">
            <xsl:for-each select="comanda/produs">
            <xsl:copy-of select="$i" />
            </xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

Efectul e acelaşi, dar elementul <i>Produs</i> nu apare direct în interiorul regulii, ci e definit la început ca o variabilă, apoi apelat din interiorul regulii. Începătorii sunt tentați să ignore faptul că variabilelor XSLT nu li se pot reatribui valori, și vor încerca să implementeze un ciclu FOR care să afișeze Produs0, Produs1, Produs2 astfel:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:variable name="i" select="0"/>
   <xsl:template match="/">
      <xsl:for-each select="comanda/produs">
         <i>Produs</i>
         <xsl:value-of select="$i"/>
         <xsl:variable name="i" select="$i+1"/>
         <br/>
      </xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

Problema e că <u>această linie</u> nu se execută niciodată, deoarece variabila $i a primit o valoare la început (0) şi ulterior nu mai poate primi valori. Rezultatul va fi afișarea repetată a lui *Produs0*.

Totuși, cum putem ajunge la rezultatul dorit? Lipsa de flexibilitate a variabilelor XSLT poate fi compensată de către interogările Xpath. Vom pasa responsabilitatea contorizării spre Xpath. Dacă vă mai amintiți din Xpath, poziția unui element între frații săi se obține prin numărarea fraților precedenți:

count(preceding-sibling::produs)
(calea e relativă la fiecare produs în parte).

Deci transformarea corectă este:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/">
      <xsl:for-each select="comanda/produs">
         <i>Produs</i>
         <xsl:value-of select="count(preceding-sibling::produs)"/>
         <br/>
      </xsl:for-each>
```

```
   </xsl:template>
</xsl:stylesheet>
```

La executarea ei, se vede că apar cuvintele *Produs0, Produs1, Produs2* unul sub altul (am concatenat și câte un BR).

Rețineți acest truc și faptul că sunt numeroase situațiile în care *lipsa de flexibilitate a variabilelor din XSLT poate fi compensată prin generarea unor valori variabile cu funcții Xpath*.

În general linia

```
<xsl:value-of select=".........."/>
```

este una din cele mai des întâlnite din XSLT. Cu ajutorul ei:

- se pot calcula valori de variabile XSLT;
- se pot extrage valori din documentul sursă, folosind Xpath.

Observație:Mai remarcați și faptul că adăugarea de conținut nou (HTML în acest caz) se face prin simpla scriere de cod nou în interiorul lui xsl:template, fără să fie necesar un operator de concatenare explicit!

Reguli de substituire recursivă

Atunci când nici Xpath nu ne ajută în a depăși problema rigidității variabilelor, mai avem o soluție la îndemână – regulile recursive.

O regulă recursivă e o regulă ce se apelează pe ea însăși, de obicei cu parametri. Mecanismul e similar cu funcțiile recursive din alte limbaje: regula XSLT se comportă ca o funcție, iar parametrii ei se comportă ca argumente ale unei funcții. **Ciclurile FOR, precum cel folosit anterior, pot fi înlocuite cu funcţii recursive care se apelează pe sine în mod repetat, de un anumit număr de ori (indicat printr-un argument al funcţiei)**.

Folosind această tehnică, problema cu numărătoarea produselor se poate rezolva și astfel:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/">
      <xsl:call-template name="generare">
         <xsl:with-param name="listaproduse" select="comanda/produs"/>
      </xsl:call-template>
   </xsl:template>

   <xsl:template name="generare">
      <xsl:param name="i" select="0"/>
      <xsl:param name="listaproduse"/>
      <xsl:if test="$listaproduse">
            <i>Produs<xsl:value-of select="$i"/></i>
            <xsl:call-template name="generare">
               <xsl:with-param name="i" select="$i+1"/>
               <xsl:with-param name="listaproduse" select="$listaproduse[position()>1]"/>
            </xsl:call-template>
      </xsl:if>
   </xsl:template>
</xsl:stylesheet>
```

Observați că acum avem 2 reguli:

Prima regulă e "programul principal" și are rolul de a aplica substituirea integrală a documentului iniţial (match="/") prin apelarea funcţiei "generare" (xsl:call-template) cu un parametru (xsl:with-param) ce

reprezintă lista produselor găsite în documentul sursă. În programarea clasică, prima regulă s-ar traduce prin construcții de forma (sintaxa e un pseudocod improvizat):

listaproduse=<vectorul produselor din comanda>
generare(listaproduse)

A doua regulă e funcţia recursivă. O recunoaştem prin faptul că nu are match, dar are nume ("generare") şi argumente (declarate în interior cu xsl:param). Funcţia se apelează pe sine în mod repetat (din nou xsl:call-template), dar de fiecare dată cu o listă de produse mai scurtă cu 1 (obţinută prin listaproduse[position()>1]) şi cu un contor $i care se măreşte la fiecare apel.

La fiecare apel se generează cuvântul Produs, urmat de contorul $i. Condiția de continuare a recursivității e <xsl:if test="$listaproduse">, adică oprim ciclul atunci când vectorul listaproduse rămâne gol (s-au terminat de eliminat produsele din el).

În programarea clasică a doua regulă s-ar traduce prin următorul pseudocod:

```
function generare(listaproduse,i=0)
{
if (listaproduse)
        {
        write "Produs"+i
        i=i+1
        listaproduse=<vectorul listaproduse fără primul element>
        generare(listaproduse,i)
        }
}
```

Observați diferența între modul de lucru al variabilelor și modul de lucru al parametrilor. Parametrii își pot schimba valoarea la apelul unei "funcții" deci nu sunt la fel de rigizi ca variabilele! De aceea **putem simula cicluri FOR prin funcții care se apelează pe ele însele folosind ca argument un contor $i care crește la fiecare apel**.

Reguli de substituire alternativă

Până aici am generat un document nou *fără să folosim conținutul existent în cel original*. Rolul uzual al unei transformări XSLT este să folosească într-un fel sau altul documentul sursă pentru a produce un document nou, ce conţine unele informaţii din sursă. În următorul exemplu, în locul cuvintelor *Produs0, Produs1, Produs2*, vom genera denumirile produselor.

Această sarcină e puțin complicată de faptul că documentul original nu are o structură regulată pentru produse! Primul produs are denumirea într-un atribut, al doilea îl are în conținutul textual, al treilea îl are într-un element-fiu. De aceea în trecut am precizat că, deşi documentele XML permit astfel de structuri neregulate, se recomandă definirea unui vocabular (cu DTD sau XML Schema) care să impună o anumită structură pe toate elementele (reţineţi: performanţa cea mai mare e când toate informaţiile sunt sub formă de atribute!)

Varianta 1.
Când avem o structură regulată (adică toate produsele arată la fel), un ciclu FOR sau o funcţie recursivă ca mai sus rezolvă problema eficient. În acest caz însă, în ciclul FOR trebuie să includem un test CASE care să verifice mai întâi de unde putem extrage denumirile produselor:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
   version="1.0">
   <xsl:template match="/">
      <xsl:for-each select="comanda/produs">
            <xsl:choose>
            <xsl:when test="@denumire">
                        <xsl:value-of select="@denumire"/>
            </xsl:when>
            <xsl:when test="denumire">
                        <xsl:value-of select="denumire"/>
            </xsl:when>
            <xsl:otherwise>
                        <xsl:value-of select="."/>
            </xsl:otherwise>
      </xsl:choose>
      <br/>
      </xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

Structura CASE e definită cu **xsl:choose**. Aceasta conține o serie de "variante" posibile, fiecare definită de câte o condiţie Xpath (atributul **test**). Ultima variantă este **xsl:otherwise** (pentru produsele la care nici una din condiţii nu se îndeplineşte). În fiecare variantă, cu xsl:value-of se extrage valoarea ce va fi inclusă în documentul final:

- Dacă găsește un atribut **denumire** (test="@denumire") i se extrage valoarea (cu **xsl:value-of**);
- Dacă găsește un element **denumire** (test="denumire") i se extrage valoarea;
- Dacă nu găsește nici una din variante (xsl:otherwise), se va considera că denumirea se află în conținutul textual (în Xpath "." reprezintă valoarea din nodul curent).

Observați că toate căile Xpath din atributele **test** și select sunt RELATIVE la nodul curent (adică produsul la care s-a ajuns cu **xsl:for-each**).

Rețineți că în general, XSLT folosește căi RELATIVE la nodul la care s-a ajuns cu procesarea:

- **căile din xsl:for-each sunt relative la nodul selectat de match**
- **căile din xsl:value-of sunt relative la nodurile parcurse cu xsl:for-each**

Varianta 2.

Structura CASE nu e singura metodă prin care putem extrage cele 3 denumiri! După cum am văzut la exemplul cu numerotarea, XSLT conlucrează cu Xpath într-o manieră flexibilă – unele sarcini pot fi transferate spre Xpath, inclusiv filtrarea de elemente. Putem să transferăm responsabilitatea extragerii denumirilor Xpath, caz în care nu mai avem nevoie de structura CASE!

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/">
      <xsl:for-each select="comanda/produs">
        <xsl:value-of  select="self::node()[not(denumire) and not(@denumire)]|denumire|@denumire"/>
<br/>
      </xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

Observați cum am evitat structura CASE mutând decizia spre Xpath:

- **denumire** returnează valoarea fiului denumire (dacă există);
- **@denumire** returnează valoarea atributului denumire (dacă există).
- **self::node()[not(denumire) and not(@denumire)]** returnează valoarea nodului curent cu condiția ca acesta să nu aibă un fiu denumire și nici un atribut denumire;

Între cele 3 variante am pus operatorul Xpath | care aplică un SAU logic între cele 3 situații!

Varianta 3.

O a treia variantă ar fi să evităm cu totul ciclul FOR, ceea ce nu e recomandat dacă numărul de elemente procesate e mare. Aici însă e suficient de mic încât să le putem interoga individual, presupunând că e cunoscută structura fiecărui produs (ceea ce, iarăşi, nu e un caz realist):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/">
      <xsl:value-of select="comanda/produs[1]/@denumire"/>
      <br/>
      <xsl:value-of select="comanda/produs[2]"/>
      <br/>
      <xsl:value-of select="comanda/produs[3]/denumire"/>
      <br/>
    </xsl:template>
</xsl:stylesheet>
```

Varianta 4.

A patra variantă mută iarăși o parte din sarcini spre Xpath: de data asta exploatăm atributul **match** al regulii. În loc să substituim tot documentul sursă, vom substitui doar elementele **produs** cu denumirile lor, apoi restul nodurilor le ştergem (prin substituire cu șirul vid).

Primul aspect e rezolvat astfel (observați cum match nu mai selectează rădăcina, ci produsele):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/comanda/produs">
   <xsl:value-of
select="self::node()[not(denumire) and not(@denumire)]|denumire|@denumire"/>
   <br/>
   </xsl:template>
</xsl:stylesheet>
```

Dacă rulați această transformare, veți observa:

- Elementele **produs** se substituie cu denumirile lor;
- Din restul documentului sursă se păstrează doar conţinutul textual (nodurile de tip text)! Aceasta e regula implicită din XSLT: **orice nod care nu e procesat de nici o regulă de substituire va fi eliminat, cu excepţia nodurilor text!**

Deci mai trebuie să punem o regulă care să elimine și aceste noduri text:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/comanda/produs">
   <xsl:value-of
      select="self::node()[not(denumire) and not(@denumire)]|denumire|@denumire"/>
   <hr/>
   </xsl:template>
   <xsl:template match="text()"/>
</xsl:stylesheet>
```

A doua regulă se aplică tuturor nodurilor de tip text (match="text()") și, fiind lipsită de conţinut, produce şirul vid (deci le șterge).

Generarea unei pagini Web complete

Dacă percepem documentul XML ca pe o bază de date, atunci interogările Xpath vor juca rolul lui SQL, iar transformările XSLT vor juca rolul limbajului PHP (pentru generare de pagini HTML din date stocate pe

server). Modificăm în continuare exemplul anterior pentru a demonstra acest principiu: vom genera din documentul sursă o pagină HTML ce prezintă produsele în formă tabelară.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="/">
    <html>
    <head> <style> #cap {color:red}  </style></head>
    <body>
    <h1>Lista produselor din comanda:</h1>
    <table border="1">
      <tr id="cap"><td>Denumire</td><td>Pret</td></tr>
      <xsl:for-each select="comanda/produs">
      <tr>
        <td>
          <xsl:value-of
            select="self::node()[not(denumire) and not(@denumire)]|denumire|@denumire"/>
        </td>
        <td>
          <xsl:value-of select="pret|@pret"/>
        </td>
      </tr>
      </xsl:for-each>
      <tr>
        <td colspan="2" align="right">
          <xsl:value-of select="sum(//pret|//@pret)"/>
        </td>
      </tr>
    </table>
    </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Rezultatul generat arată astfel:

Lista produselor din comanda

Denumire	Pret
Televizor	100
Calculator	200
Ipod	300
	600

Figura 21 Generarea unei pagini html prin trasformare XSLT

Observație: S-a generat o pagină Web completă, cu tot cu CSS (ar putea conține chiar și JavaScript cu cod AJAX!) Din acest exemplu reiese posibilitatea de a folosi XSLT ca generator dinamic de pagini Web, înlocuind în acest sens limbajul PHP!

Observație: Totalul nu a fost extras din documentul sursă, deşi era deja disponibil acolo! (în elementul <prettotal>). Aici a fost recalculat cu ajutorul Xpath. În cazuri realiste, valorile calculate (precum totalurile) nu se stochează în documentul XML original, ci se calculează în mod dinamic ca în acest exemplu.

Regula de conservare

S-a văzut la început că XSLT nu conservă implicit conţinutul XML original. Dacă dorim să se conserve anumite elemente, trebuie să creăm o regulă de conservare care să substituie elementele cu ele însele. Această regulă va include **xsl:copy-of** pe interogarea ce returnează nodul curent (pentru execuție, modificați scenariul de transformare să nu se mai returneze html în browser):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:template match="/">
      <xsl:copy-of select="."/>
   </xsl:template>
</xsl:stylesheet>
```

Această regulă substituie elementul rădăcină cu el însuși, producând o copie a documentului original. Următorul exemplu produce un rezultat similar, dar elimină nodurile invizibile cu xsl:strip-space:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:strip-space elements="*"/>
   <xsl:template match="/">
      <xsl:copy-of select="."/>
   </xsl:template>
</xsl:stylesheet>
```

Următorul exemplu conservă toate elementele produs (selectate cu match), plus nodurile text din afara produselor (care se conservă în mod implicit):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/comanda/produs">
      <xsl:copy-of select="."/>
   </xsl:template>

</xsl:stylesheet>
```

În următorul exemplu am folosit xsl:copy în loc de xsl:copy-of. Diferența e că se conservă produsele FĂRĂ conținut și atribute! Această tehnică se folosește când vrem să conservăm elemente XML, dar dorim să le redefinim conținutul și atributele:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/comanda/produs">
      <xsl:copy/>
   </xsl:template>

</xsl:stylesheet>
```

Următorul exemplu e similar dar, profitând de faptul că xsl:copy nu conservă atribute şi conţinut, vom crea unele noi:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/comanda/produs">
```

```
      <xsl:copy>
         <xsl:attribute name="NouAtribut">NouaValoare</xsl:attribute>
<NouFiu>Nou continut</NouFiu>
</xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

O altă variantă ar fi ca în loc de xsl:copy să retastăm elementul original, cu modificările dorite:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/comanda/produs">
      <produs PretNou="{@pret|pret}">
         <NouFiu>Nou continut</NouFiu>
      </produs>
   </xsl:template>

</xsl:stylesheet>
```

Observați cum am creat un atribut PretNou care își ia valoarea fie din atributul vechi pret, fie din elementul pret (depinde pe care-l gasește).

În următorul exemplu se conservă produsele, apoi conținutul lor e redefinit să fie egal cu prețul (extras fie din atributul pret, fie din elementul-fiu pret, de la caz la caz).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/comanda/produs">
      <xsl:copy>
         <xsl:value-of select="@pret|pret"/>
      </xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

În următorul caz conservăm elementele produs împreună cu conținutul lor textual.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:template match="/comanda/produs">
      <xsl:copy>
         <xsl:apply-templates/>
      </xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

Instrucțiunea xsl:apply-templates se utilizează împreună cu xsl:copy și are rolul ca, după ce s-a făcut o copie goală a elementului, să se umple cu rezultatul eventualelor reguli **care mai există pentru fiii elementului**. În cazul de față, neexistând alte reguli, se aplică regula implicită (conservarea nodurilor de tip text), deci obținem produsele fără atribute dar păstrându-le conținutul textual.

În următorul exemplu înlocuim regula implicită cu una precizată de noi:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="/comanda/produs">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="produs/*|produs/text()[normalize-space(.)!='']">
    Aici a fost candva un fiu (vizibil)
  </xsl:template>

</xsl:stylesheet>
```

De data aceasta, xsl:apply-templates caută dacă există reguli pentru fiii produselor și găsește una:

- Ea se aplică fiilor de tip element (*) și fiilor de tip text vizibil (text()[normalize-space(.)!=''])[10]
- Conform acesteia, toți fiii unui element produs vor fi înlocuiți cu textul din interiorul regulii ("Aici a fost...").

Mai departe aplicăm o serie de reguli în cascadă:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="/comanda">
    <xsl:copy>
      <xsl:attribute name="client">Pop Ion</xsl:attribute>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="produs">
    <xsl:copy>
      <xsl:attribute name="PretNou">
        <xsl:value-of select="@pret|pret"/>
      </xsl:attribute>
      <ID><xsl:value-of select="@id|id"/></ID>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="text()[normalize-space(.)!='']"/>

</xsl:stylesheet>
```

Prima regulă copiază (cu xsl:copy) o versiune goală a rădăcinii comanda, îi adaugă atributul client="Pop Ion" (cu xsl:attribute) și o umple cu rezultatul eventualelor reguli care există pentru fiii comenzii.

Astfel se ajunge la **a doua regulă** care:

- produce versiuni goale ale produselor (cu xsl:copy)
- le creează atributul PretNou (cu xsl:attribute) cu valoarea luată din prețurile existente (fie din atributul vechi pret, fie din elementul pret)
- le adaugă elemente-fiu cu numele **ID** şi valoarea luată din id-urile vechi (atribute sau elemente, după caz)
- și adaugă mai departe rezultatul eventualelor reguli care mai există pentru fiii produselor.

[10] Sunt două apostroafe, nu ghilimele, pentru compararea unui string cu şirul vid!

Astfel se ajunge la **a treia regulă** care caută noduri text vizibile și le șterge. A treia regulă e apelată atât de prima regulă (ștergând nodurile 600 și "Aceasta este o comanda" care s-ar păstra datorită regulii implicite de conservare a textului) cât și de a doua regulă (ștergând textul vechi din fiecare produs, care s-ar conserva datorită aceleiași reguli implicite). Rezultatul arată astfel (nodurile invizibile nu s-au eliminat!):

```
<?xml version="1.0" encoding="utf-8"?><comanda client="Pop Ion">

    <produs PretNou="100"><ID>p1</ID></produs>
    <produs PretNou="200"><ID>p2</ID></produs>
    <produs PretNou="300"><ID>p3</ID>


    </produs>
    </comanda>
```

Figura 22 Rezultatul aplicării transformării XSLT

Rețineți regulile implicite din XSLT:

- Din documentul sursă se păstrează în mod implicit doar nodurile text.
  - Restul informaţiei (elemente, atribute) au nevoie de reguli explicite pentru a fi conservate;
  - Dacă dorim eliminarea nodurilor text e nevoie de reguli explicite de ştergere a lor;
- Toate nodurile din documentul original care NU intră sub incidența NICIUNEI reguli de substituire se supun regulii implicite (eliminare, cu excepția nodurilor text);
- Dacă un același nod e afectat de mai multe reguli de substituire, i se aplică regula cu match-ul cel mai specific (ex: match="*comanda/produs*" are prioritate față de match="*produs*", care are prioritate faţă match="*node()*").

Cum putem rula transformări XSLT fără Oxygen?

Putem aplica XSLT direct pe server, inclusiv în PHP, astfel încât să livrăm spre JavaScript direct rezultatul transformării. Transformarea XSLT în PHP e frecvent folosită când codul XML a sosit în PHP din surse externe (de la un serviciu Web, un depozit de documente XML).
Salvaţi sursa în fişierul sursa.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="transf.xsl"?>
<Comanda Client="PopIon">
  <Produs ID="P1" Pret="100">Televizor</Produs>
  <Produs ID="P2" Pret="30">Ipod</Produs>
</Comanda>
```

Salvaţi transformarea în fişierul transformare.xsl:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="/">
    <h1>Comanda lui <xsl:value-of select="Comanda/@Client"/></h1>
    <table border="1">
      <tr>
        <td>Denumire</td>
        <td>Pret</td>
      </tr>
      <xsl:for-each select="Comanda/Produs">
        <tr>
          <td><xsl:value-of select="."/></td>
          <td><xsl:value-of select="@Pret"/></td>
        </tr>
      </xsl:for-each>
      <tr>
```

```
            <td colspan="2" align="left">
              Total:
              <xsl:value-of select="sum(Comanda/Produs/@Pret)"/>
            </td>
          </tr>
        </table>
    </xsl:template>
</xsl:stylesheet>
```

Creaţi scriptul PHP cu codul ce execută transformarea:

```
<?php
$sursa = new DOMDocument();
$sursa->load('sursa.xml');
$transformarea = new DOMDocument();
$transformarea->load('transformare.xsl');
$convertor = new XSLTProcessor();
$convertor->importStyleSheet($transformarea);
print $convertor->transformToXML($sursa);
?>
```

Executaţi exemplul accesând scriptul PHP prin localhost, şi browserul ar trebui să afişeze direct rezultatul transformării fără a necesita altă acţiune din partea utilizatorului.

Dacă la execuția scriptului php apare mesajul de eroare “**Fatal error**: Class 'XSLTProcessor' not found ...”, în fișierul “php.ini” din xampp\php căutați linia “;extension=php_xsl.dll” și ștergeți caracterul ; ce o precede, după care reporniți serverul Apache.

2. Interoperabilitate sintactică bazată pe JSON și cereri AJAX cross-domain

Pentru a avea interoperabilitate între aplicații nu e suficient să stăpânim structurile de date ce se pot transmite, ci și mecanismul de comunicare a lor, adică protocolul HTTP conform căruia vom programa cereri de trimitere sau solicitare de date. În exercițiile următoare vom analiza mai multe moduri de a utiliza protocolul HTTP pentru a schimba date între un client AJAX (JavaScript) domenii diferite. În plus, vom folosi structuri de date JSON[11] – o alternativă mai simplă la XML care oferă performanță superioară la transferul de date, însă nu beneficiază de toată gama de tehnologii discutată în capitolul precedent.

2.1 Crearea de domenii virtuale multiple în aceeaşi instalare Apache

Pentru a simula cereri cross-domain AJAX[12] vom crea 2 servere virtuale în XAMPP[13], care în loc să fie accesate prin localhost, vor putea fi accesate prin adrese de domeniu fictive alese de noi – în acest caz adresele vor fi **site1.com** şi **site2.com**. Pașii de mai jos descriu modul de definire a acestor domenii fictive în Windows 7 (alte sisteme de operare vor stoca fișierele indicate în alte locații):

Pas1. Creaţi în htdocs câte un folder corespunzător fiecăruia dintre cele două site-uri: site1 şi site2

Pas2. Creaţi în fiecare din aceste foldere câte un script de test (cu numele script1.php şi script2.php), care să indice vizitatorului printr-un mesaj simplu pe care site se află:

```
<?php
print "acesta e site-ul x";
?>
```

(în loc de x puneţi 1 în script1.php, salvat în folderul site1 şi puneţi 2 în script2.php, salvat în folderul site2)

Pas3. Asiguraţi-vă că toate componentele XAMPP-ului sunt oprite.

Pas4. Intraţi în folderul de instalare XAMPP până la fişierul httpd-vhosts.conf (la o instalare normală îl veţi găsi în folderul C:\xampp\apache\conf\extra)

Pas5. În interiorul acestui fişier putem defini multiple domenii fictive, fiecare dintre ele cu o altă adresă, alte setări şi alt folder principal (în loc de htdocs). În mod implicit, tot conţinutul fişierului e dezactivat prin comentarii. Activaţi linia care începe cu NameVirtualHost şi adăugaţi la final domeniile fictive care doriţi să devină accesibile prin browser. Fişierul final ar trebui să arate astfel (am înlocuit parte din liniile comentate cu puncte de suspensie):

```
# Virtual Hosts
#..............
# Use name-based virtual hosting.
#
NameVirtualHost *:80
#
# VirtualHost example:
# .....
##</VirtualHost>

<VirtualHost *:80>
   ServerAdmin webmaster@localhost
   DocumentRoot "C:/xampp/htdocs"
   ServerName localhost
```

[11] https://www.json.org/

[12] https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

[13] https://www.apachefriends.org/

```
	<Directory "C:/xampp/htdocs">
		Options Indexes FollowSymLinks Includes ExecCGI
		AllowOverride All
		Order allow,deny
		Allow from all
	</Directory>
</VirtualHost>

<VirtualHost *:80>
   ServerAdmin webmaster@site1.com
   DocumentRoot "C:/xampp/htdocs/site1"
   ServerName site1.com
   ErrorLog "logs/site1.com-error.log"
   CustomLog "logs/site1.com-access.log" common

	<Directory "C:/xampp/htdocs/site1">
		Options Indexes FollowSymLinks Includes ExecCGI
		AllowOverride All
		Order allow,deny
		Allow from all
	</Directory>
</VirtualHost>

<VirtualHost *:80>
   ServerAdmin webmaster@site2.com
   DocumentRoot "C:/xampp/htdocs/site2"
   ServerName site2.com
   ErrorLog "logs/site2.com-error.log"
   CustomLog "logs/site2.com-access.log" common

	<Directory "C:/xampp/htdocs/site2">
		Options Indexes FollowSymLinks Includes ExecCGI
		AllowOverride All
		Order allow,deny
		Allow from all
	</Directory>
</VirtualHost>
```

Observaţii:

- Fiecare domeniu fictiv se creează cu un element <VirtualHost> ce conţine setările domeniului. Setările cheie sunt **DocumentRoot** (folderul în care se stochează fişierele site-ului) şi **ServerName** (adresa de domeniu prin care se va accesa site-ul) şi marcatorul **Directory** (în care se indică din nou folderul, de data asta împreună cu diverse drepturi de acces)
- În acest exemplu am definit 3 domenii: **localhost** (cel implicit, ce va folosi fişiere din htdocs), **site1.com** (ce va folosi fişiere din htdocs/site1) şi **site2.com** (ce va folosi fişiere din htdocs/site2)

Pas6. Aceste site-uri trebuie definite şi la nivelul sistemului de operare, într-un fişier numit hosts. În Windows 7 acest fişier poate fi găsit în C:\Windows\System32\drivers\etc

Pas7. Editarea fişierului necesită drepturi de administrator. De exemplu dacă îl editaţi cu Notepad++[14], NU daţi click dreapta pe fişier urmat de *Open with Notepad++* (la salvare veţi primi eroare). Trebuie să porniţi mai întâi editorul cu click dreapta pe Notepad++ - *Run as administrator* şi să deschideţi fişierul din editor. Adăugaţi la finalul fişierului câte o mapare între adresa IP pentru localhost (127.0.0.1) şi cele două adrese de domeniu fictive:

```
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#......................................................
# localhost name resolution is handled within DNS itself.
#	127.0.0.1	localhost
```

[14] https://notepad-plus-plus.org/

```
#		::1		localhost
127.0.0.1	site1.com
127.0.0.1	site2.com
```

Pas8. Porniţi Apache-ul din XAMPP. Dacă scripturile de la Pas2 au fost corect salvate, ar trebui să le puteţi accesa prin adresele:
http://site1.com/script1.php
http://site2.com/script2.php

În plus, adresa http://localhost ar trebui să funcţioneze în continuare normal, permiţând acces la fişierele din htdocs.

2.2 Crearea de cereri HTTP prin XMLHttpRequest

Mai întâi creaţi o cerere HTTP simplă spre un script PHP care returnează o structură de date JSON, în cadrul folderului site1. Construiţi mai întâi scriptul PHP (salvat cu numele datejson.php):

```
<?php
header("Content-type:application/json");
$Produs1=array("ID"=>"P1","Pret"=>100,"Denumire"=>"Televizor");
$Produs2=array("ID"=>"P2","Pret"=>30,"Denumire"=>"Ipod");
$Produse=array($Produs1,$Produs2);
$Raspuns=array("Comanda"=>array("Client"=>"Poplon","Produse"=>$Produse));
print json_encode($Raspuns);
?>
```

Observaţii:

- Atunci când scripturile PHP returnează altceva decât cod HTML, funcţia header de pe prima linie se foloseşte pentru a declara tipul conţinutului. Declaraţia e stocată în câmpul de antet HTTP Content-type, a cărui valoarea trebuie să fie codul MIME al formatului returnat (pentru formatul JSON codul MIME este application/json)[15];
- Codul JSON e obţinut prin construirea unui vector asociativ PHP cu funcţia array(); la final, vectorul complet e convertit în şir de caractere JSON cu json_encode[16].

Accesaţi scriptul (site1.com/datejson.php) pentru a vedea forma în care ajung datele la browser:

```
{"Comanda":{"Client":"Poplon","Produse":[{"ID":"P1","Pret":100,"Denumire":"Televizor"},{"ID":"P2","Pret":30,"Denumire":"Ipod"}]}}
```

De acest format trebuie să ţinem cont în pagina client care va extrage informaţiile. Construiţi mai departe pagina client – ea va consta într-un singur buton al cărui apăsare declanşează cererea HTTP, obţine datele JSON şi extrage din ele denumirea primului produs:

```
<html>
<head>
<script>
function solicitare()
		{
		cerere=new XMLHttpRequest()
		cerere.onreadystatechange=procesareRaspuns
		cerere.open("GET","datejson.php")
		cerere.send(null)
		}
function procesareRaspuns()
		{
		if (cerere.readyState==4)
```

[15] Pentru majoritatea browserelor formatul JSON e recunoscut fără să trebuiască declarat dar e bine să ne obişnuim să folosim funcţia header() în scripturi ce nu returnează pagini HTML
[16] Pentru lucrul cu JSON în PHP consultaţi http://php.net/manual/ro/book.json.php (veţi remarca şi funcţia inversă json_decode() ce creează un obiect sau vector PHP din stringuri JSON)

```
                    if (cerere.status==200)
                    {
                    raspuns=JSON.parse(cerere.responseText)
                    produs=raspuns.Comanda.Produse[0].Denumire
// în loc de linia de deasupra, puteţi extrage date JSON şi cu sintaxa vectorială:
// produs=raspuns["Comanda"]["Produse"][0]["Denumire"]
                    tinta=document.getElementById("tinta")
                    tinta.innerHTML+=produs
                    }
          }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

Observaţii:

- Corpul paginii conţine doar butonul ce declanşează cererea prin onclick şi un DIV gol în care se va insera valoarea extrasă din pachetul JSON venit de la server;
- Cererea e declanşată cu ajutorul clasei XMLHttpRequest() prin metoda GET; la sosire, răspunsul e procesat cu funcţia procesareRaspuns();
- Funcţia de procesare a răspunsului testează succesul cererii (status 200 şi readyState 4), apoi preia datele din responseText şi le converteşte într-un obiect JavaScript cu JSON.parse();
- După conversie, denumirea primului produs poate fi extrasă fie prin sintaxa obiectuală (raspuns.Comanda.Produse[0].Denumire), fie prin cea vectorială (raspuns["Comanda"]["Produse"][0]["Denumire"]); a doua variantă e preferabilă atunci când etichetele din JSON sunt mai complexe, conţinând şi caractere ce pot crea probleme în denumiri de atribute obiectuale (de exemplu în loc de *raspuns.Comanda Mea* din cauza spaţiului se va prefera *raspuns["Comanda Mea"]*);
- După extragerea denumirii produsului, aceasta e injectată în DIV-ul gol pregătit în acest sens; DIV-ul s-a selectat cu getElementById iar conţinutul său e modificat cu ajutorul innerHTML.

Testaţi pagina în browser – la apăsarea butonului ar trebui să se adauge sub butonul denumirea primului produs.

Mai departe, mutaţi fişierul datejson.php din folderul în care a fost creat (site1) în folderul corespunzător celuilalt site fictiv (site2). În exemplul de mai sus, modificaţi linia prin care se declară metoda şi adresa cererii HTTP astfel încât browserul să aibă impresia că se accesează un alt server:

```
cerere.open("GET","http://site2.com/datejson.php")
```

Testaţi din nou pagina şi veţi remarca faptul că nu mai funcţionează (consola browserului va indica o eroare de origine). Aceasta se datorează restricţiei implicite ca obiectele XMLHttpRequest să nu poată solicita date de pe alte site-uri (domenii Web) decât cel în care se află şi pagina HTML ce iniţiază cererea.

Aceasta era o restricţie utilă înainte de apariţia serviciilor Web (numite în unele contexte şi Web APIs sau REST APIs). Serviciile Web au fost concepute pentru a putea fi accesate de către orice clienţi care au nevoie de ceea ce oferă serviciul (de obicei nişte date structurate în cod XML sau JSON) aşa că restricţia amintită a trebuit să poată fi depăşită. Exemplificăm în continuare câteva tehnici în acest sens.

2.3 Crearea de cereri HTTP cross-domain prin tehnica JSON-P

O posibilitate de a realiza cereri HTTP între domenii diferite (numite şi "cross-domain AJAX calls") este folosirea unui marcator SCRIPT dinamic. Noua versiune a paginii client arată astfel:

```
<html>
<head>
```

```
<script type="application/javascript" id="bloc"></script>
<script>
function solicitare()
		{
		blocJS=document.getElementById("bloc")
		blocJS.src="http://site2.com/datejson.php"
		}
function procesareRaspuns(date)
		{
		produs=date.Comanda.Produse[0].Denumire
		tinta=document.getElementById("tinta")
		tinta.innerHTML+=produs
		}
</script>

</head>

<body>
		<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
		<div id="tinta"></div>

</body>
</html>
```

Observaţi:

- Pagina are la început un marcator SCRIPT care are doar un ID şi nu conţine deloc cod JavaScript. Marcatorii SCRIPT goi se folosesc de obicei împreună cu atributul SRC pentru a importa un script dintr-un fişier extern (de exemplu când se folosesc librării precum Jquery). În cazul de faţă, şi acel SRC lipseşte!
- Atributul SRC este creat de funcţia solicitare(). Lui se atribuie adresa scriptului PHP ce oferă datele JSON. Remarcaţi că, spre deosebire de XMLHttpRequest, atributul SRC permite accesarea de fişiere de pe alte site-uri (amintiţi-vă şi IMG SRC sau IFRAME SRC care pot accesa resurse externe);
- Crearea atributului SRC declanşează automat accesarea scriptului PHP, aducerea informaţiei returnate de acesta şi executarea sa. Ca acest lucru să fie posibil, informaţia returnată de PHP TREBUIE
    - o să fie cod JavaScript valid (marcatorul SCRIPT aşteaptă cod JavaScript!)
    - o acel cod JavaScript să indice ce funcţie din pagina client să se declanşeze imediat după ce răspunsul soseşte! Cu alte cuvinte trebuie modificat scriptul PHP să returneze nu doar datele JSON, ci şi funcţia din pagina client care se va ocupa de ele:

```
<?php
header("Content-type:application/javascript");
$Produs1=array("ID"=>"P1","Pret"=>100,"Denumire"=>"Televizor");
$Produs2=array("ID"=>"P2","Pret"=>30,"Denumire"=>"Ipod");
$Produse=array($Produs1,$Produs2);
$Raspuns=array("Comanda"=>array("Client"=>"PopIon","Produse"=>$Produse));
print "procesareRaspuns(".json_encode($Raspuns).")";
?>
```

Observaţi cum pe prima linie am declarat acum că scriptul returnează cod JavaScript, iar în ultima linie am concatenat funcţia în jurul datelor. Această concatenare va face ca la browser să ajungă un răspuns de forma:

```
procesareRaspuns(...date...)
```

Odată ajuns în browser, marcatorul SCRIPT va provoca executarea acestei linii de cod, beneficiind de faptul că există o funcţie cu acest nume pregătită să se ocupe de date.

Apare însă o problemă nouă: dacă presupunem că pagina client şi scriptul PHP aparţin unor organizaţii diferite, e puţin probabil ca scriptul PHP să ŞTIE cum se numeşte funcţia din pagina client. Din acest motiv, pentru scripturile server (de obicei servicii Web) care oferă această facilitate s-a stabilit o convenţie: pagina

client trebuie să anunţe serverul CUM SE NUMEŞTE funcţia de procesare a datelor, iar acest lucru trebuie făcut printr-un parametru GET denumit callback. Serverul trebuie să fie pregătit să preia din acest parametru numele funcţiei şi să-l concateneze corespunzător.

În pagina client, înlocuiţi funcţia solicitare cu următoarea variantă:

```
function solicitare()
        {
        blocJS=document.getElementById("bloc")
        blocJS.src="http://site2.com/datejson.php?callback=procesareRaspuns"
        }
```

În scriptul server, înlocuiţi ultima linie cu:

```
$NumeFunctieClient=$_GET["callback"];
print $NumeFunctieClient."(".json_encode($Raspuns).")";
```

Această convenţie şi mod de lucru poartă numele de tehnică JSON-P şi este una din cele mai populare căi de a solicita date de la servicii şi servere externe. Tehnica este totuşi limitată la cereri de tip GET (acesta e tipul de cerere pe care îl declanşează orice atribut SRC din HTML).

Mai departe, modificăm exemplul pentru a beneficia de librăria Jquery[17] ce are inclus suport implicit pentru tehnica JSON-P, scutindu-ne de o parte din paşii exemplelor anterioare.

```
<html>
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function solicitare()
        {
        adresa="http://site2.com/datejson.php?callback=?"
        $.getJSON(adresa, function(date)
                                {
                                produs=date.Comanda.Produse[0].Denumire
                                continutDeInserat="<div>"+produs+"</div>"
                                $(continutDeInserat).appendTo(document.body)
                                }
                )
        }
</script>
</head>

<body>
        <input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
</body>
</html>
```

Observaţi simplificarea codului:

- Am importat versiunea 2.2.4 a librăriei Jquery, pusă la dispoziţie de Google pentru acces dinamic. Puteţi consulta toate librăriile puse la dispoziţie de Google la adresa https://developers.google.com/speed/libraries/#jquery (unele în mai multe versiuni). Acesta e un mod convenabil de utilizare Jquery în sensul că nu mai trebuie să descărcăm librăria şi să o importăm dintr-un folder al nostru. Dezavantajul e că la execuţie trebuie să fim conectaţi la Internet.
- Adresa la care trimitem cererea se termină cu callback=?, ceea ce, în combinaţie cu utilizarea getJSON, are ca efect ACTIVAREA tehnicii JSON-P (fără ca noi să mai trebuiască să construim acel SCRIPT SRC dinamic)

[17] https://jquery.com/

- Semnul întrebării sugerează că NU mai trebuie să dăm nume funcţiei de procesare a răspunsului. Aceasta e livrată direct ca argument al funcţiei getJSON (după adresă). Acest gen de funcţii-argument poartă numele de **funcţii anonime** – adică funcţii ce se apelează într-un singur loc şi în consecinţă nu are rost să li se mai dea nume (trebuie doar definite cu function() exact acolo unde trebuie şi apelate)
- Studiaţi corpul acelei funcţii anonime. Ea realizează acelaşi lucru ca înainte, dar folosind facilităţi JQuery. Mai exact, întreg blocul DIV e generat dinamic cu funcţia $()[18], apoi inserat la finalul paginii cu appendTo(document.body) (observaţi că în BODY nu am mai rezervat acel DIV gol).

O mulţime de servicii Web pun deja la dispoziţia doritorilor tehnica JSON-P. Un exemplu de astfel de serviciu este geoNames, ce oferă acces la informaţii geografice. Înlocuiţi funcţia solicitare cu:

```
function solicitare()
	{
	adresa="http://www.geonames.org/postalCodeLookupJSON?postalcode=10504&country=US&callback=?"
	$.getJSON(adresa, function(date)
					{
					locatie=date.postalcodes[0].placeName
					continutDeInserat="<div>"+locatie+"</div>"
					$(continutDeInserat).appendTo(document.body)
					}
		)
	}
```

Funcţia returnează numele locaţiei corespunzătoare unui cod poştal (10504) şi unei ţări (US) care se trimit ca parametri GET. Observaţi şi parametrul callback=? necesar tehnicii JSON-P în JQuery.

Observaţi că denumirea locaţiei e extrasă din postalcodes[0].placeName. Cum aflăm acest lucru? Cel mai uşor este să consultăm răspunsul serviciului geoNames în consola browserului. În imaginea de mai jos aveţi consola Chrome (cu opţiunea Network se văd cererile HTTP, apoi selectăm ultima cerere iar în dreapta putem verifica datele sosite în Preview sau Response ca să aflăm că avem nevoie de postalcodes[0].placeName pentru a ajunge la denumirea locaţiei):

[18] Atenţie, funcţia $() trebuie să aibă ca argument un element HTML complet. Nu va putea insera direct produsul prin $(produs), are nevoie să împachetăm textul inserat în DIV.

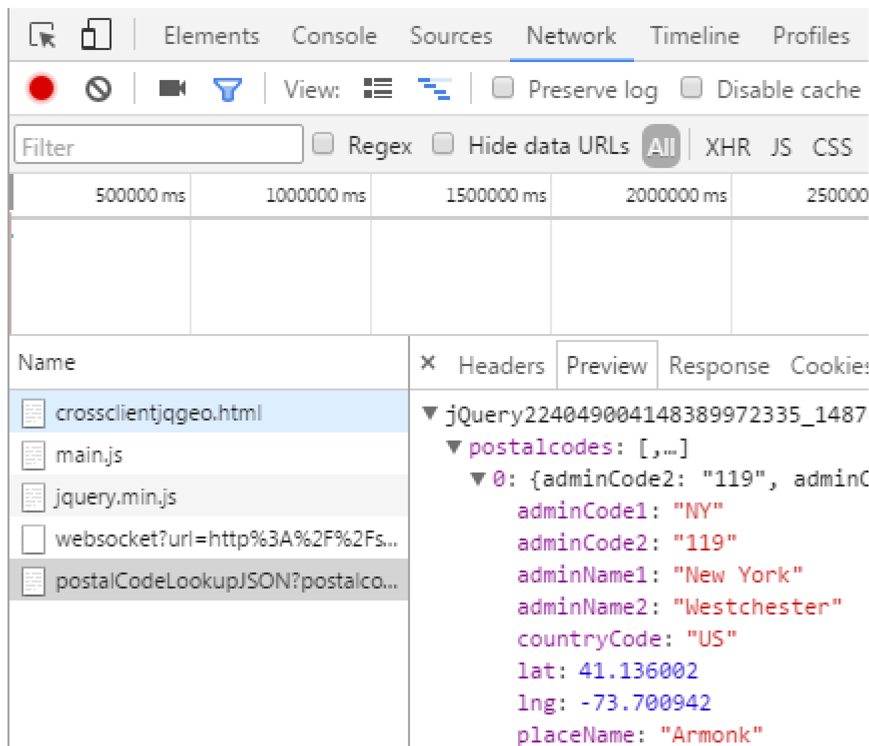


Figura 23 Vizualizarea răspunsului de la serviciul geoNames în consola browserului

Mai multe detalii despre serviciile puse la dispoziţie de geoNames, inclusiv adresele şi parametri cu care acestea se pot apela găsiţi la:
http://www.geonames.org/export/web-services.html

2.4 Crearea de cereri HTTP cross-domain prin scripturi server proxy

Tehnica JSON-P are o serie de limitări importante, cum ar fi limitarea la cereri de tip GET (deci clientul poate trimite o cantitate limitată de date sub formă de parametri GET, fără date confidenţiale, fără upload de fişiere etc.). O soluţie premergătoare tehnicii JSON-P, cu securitate mai bună dar performanţă mai slabă e cea a scriptului proxy: acesta e un script server-side de intermediere, care solicită datele dorite de la serverul remote şi le pune la dispoziţia paginii client. Aceasta se bazează pe faptul că restricţia cererilor HTTP cross-domain NU există între servere (între scripturi PHP). Este şi un principiu des folosit la comunicarea cu multiple servicii Web (numite şi *aplicaţii mash-up*, care în loc să folosească o bază de date proprie adună date de la multiple servicii punându-le la dispoziţia clientului într-o formă integrată – vezi agregatoarele ofertelor de bilete de avioane).

În tehnica scriptului proxy avem cel puţin trei fişiere: (1) scriptul remote care oferă datele prin HTTP; (2) pagina client care are nevoie de date, pe un alt server; (3) pe acelaşi server cu pagina client, un script de intermediere ce forwardează cererea clientului spre script, făcând apoi acelaşi lucru cu datele sosite.

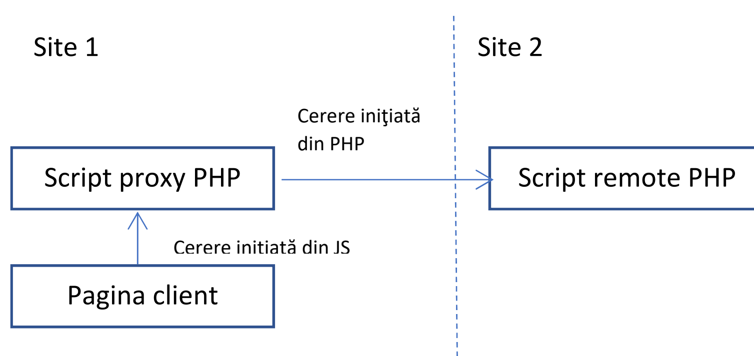


Figura 24 Cereri HTTP cross-domain prin script server proxy

Construiţi scriptul remote, în folderul site2, cu numele datejson.php:

```
<?php
header("Content-type:application/json");
$Produs1=array("ID"=>"P1","Pret"=>100,"Denumire"=>"Televizor");
$Produs2=array("ID"=>"P2","Pret"=>30,"Denumire"=>"Ipod");
$Produse=array($Produs1,$Produs2);
$Raspuns=array("Comanda"=>array("Client"=>"Poplon","Produse"=>$Produse));
print json_encode($Raspuns);
?>
```

E acelaşi cod ca în exemplele precedente, cu construirea structurii de date şi livrarea acesteia după conversie în string JSON cu json_encode.

Apoi construim scriptul proxy, în folderul site1, cu numele scriptproxy.php.

```
<?php
header("Content-type:application/json");
$cerere=curl_init();
$configurari=array(CURLOPT_RETURNTRANSFER=>1,CURLOPT_URL=>"http://site2.com/datejson.php");
curl_setopt_array($cerere,$configurari);
$rezultat=curl_exec($cerere);
print $rezultat;
curl_close($cerere);
?>
```

Observaţii:

- Aici se iniţiază o cerere HTTP între scripturi PHP. Spre deosebire de cererile de până acum, care erau iniţiate din JavaScript (prin XMLHttpRequest, prin JQuery sau prin SRC), în PHP acestea se construiesc cu ajutorul instrumentului cURL şi o sintaxă specifică PHP, urmând următorii paşi:
  - Se creează cererea (curl_init());
  - Se configurează cererea cu ajutorul unui vector de variabile predefinite: în CURLOPT_URL se defineşte destinaţia cererii (adresa scriptului remote), în CURLOPT_RETURNTRANSFER valoarea 1 stabileşte ca răspunsul să vină sub forma unui string[19].
  - Se defineşte cererea configurată cu curl_setopt_array();
  - Se execută cererea cu curl_exec() preluând răspunsul într-o variabilă;
  - Se distruge cererea cu curl_close().
- Cu print, răspunsul e returnat mai departe paginii client care are nevoie de el.

cURL nu e singurul instrument disponibil în PHP pentru a executa cereri HTTP – se mai pot folosi funcţii care pot citi direct informaţia dintr-un fişier accesibil prin URL - vezi file_get_contents()[20] sau fopen()[21]. Preferăm totuşi cURL deoarece pune la dispoziţie numeroase configurări posibile cu ajutorul acelor variabile predefinite (care încep cu CURLOPT). În plus, cURL este o librărie care există şi în afara PHP, putând fi folosită cu configurări similare şi în alte limbaje, şi chiar ca program executabil de sine stătător (dacă vrem să iniţiem cereri HTTP din linia de comandă a sistemului de operare – vezi https://curl.haxx.se/).

Ultimul pas este să creăm pagina client care solicită datele de la scriptul proxy, stocată pe acelaşi site cu acesta (site1). Folosim pagina iniţială, cu cererea trimisă prin XMLHttpRequest. Această pagină nu va şti de la ce server vin datele, deoarece ea comunică doar cu scriptul proxy care se află pe acelaşi server. Scriptul proxy e responsabil cu culegerea datelor din surse externe, eventual şi combinarea mai multor surse şi pregătirea datelor într-o structură integrată.

---

[19] Toate variabilele predefinite ce se pot folosi la configurarea unei cereri pot fi consultate la http://php.net/manual/en/function.curl-setopt.php, În timp ce documentaţia completă pentru cURL e disponibilă la http://php.net/manual/en/book.curl.php

[20] http://php.net/manual/ro/function.file-get-contents.php

[21] http://php.net/manual/ro/function.fopen.php

```
<html>
<head>
<script>
function solicitare()
        {
        cerere=new XMLHttpRequest()
        cerere.onreadystatechange=procesareRaspuns
        cerere.open("GET","scriptproxy.php")
        cerere.send(null)
        }
function procesareRaspuns()
        {
        if (cerere.readyState==4)
                if (cerere.status==200)
                {
                raspuns=JSON.parse(cerere.responseText)
                produs=raspuns.Comanda.Produse[0].Denumire
                tinta=document.getElementById("tinta")
                tinta.innerHTML+=produs
                }
        }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

În continuare refacem exemplul într-o nouă variantă: scriptul proxy nu va mai returna întreaga structură JSON, ci se va ocupa şi cu extragerea denumirii produsului de afişat. În consecinţă va oferi clientului doar informaţia necesară afişării, preluând o parte din efortul de procesare a răspunsului. Pentru aceasta însă, clientul trebuie să îi comunice scriptului proxy de care produs are nevoie. Pentru aceasta va transmite prin cererea HTTP un identificator de produs primind în schimb denumirea sa.

```
<html>
<head>
<script>
function solicitare()
        {
        cerere=new XMLHttpRequest()
        cerere.onreadystatechange=procesareRaspuns
        cerere.open("GET","scriptproxy2.php?identificator=P1")
        cerere.send(null)
        }
function procesareRaspuns()
        {
        if (cerere.readyState==4)
                if (cerere.status==200)
                {
                tinta=document.getElementById("tinta")
                tinta.innerHTML+=cerere.responseText
                }
        }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

Observaţi că funcţia de procesare a răspunsului nu mai realizează JSON.parse, ci se bazează pe faptul că răspunsul de la server e chiar denumirea ce se poate injecta direct în pagină. Observaţi şi că adresa la care se trimite cererea are ataşat identificatorul produsului dorit sub formă de parametru GET.

Creaţi acum noua versiune a scriptului proxy – scriptproxy2.php în folderul site1:

```
<?php
$cerere=curl_init();
$configurari=array(CURLOPT_RETURNTRANSFER=>1,CURLOPT_URL=>"http://site2.com/datejson.php");
curl_setopt_array($cerere,$configurari);
$rezultat=curl_exec($cerere);
$date=json_decode($rezultat);
$identificator=$_GET["identificator"];
$produse=$date->Comanda->Produse;
foreach ($produse as $produs)
		{
		if ($produs->ID==$identificator) {$denumire=$produs->Denumire; break;}
		}
print $denumire;
curl_close($cerere);
?>
```

Observaţi structura care preia identificatorul, îl caută în structura JSON sosită şi pe baza sa extrage doar denumirea pe care o pune la dispoziţie, ca valoare simplă, paginii client.
În scriptul remote nu se modifică nimic, acesta oferă pachetul JSON complet ca înainte.

În continuare refacem exemplul într-o a treia variantă, în care toate cererile folosesc metoda POST în loc de GET (ceea ce nu ar fi posibil cu tehnica JSON-P). Mai mult, identificatorul va fi împins până la scriptul remote, şi acesta va returna doar denumirea solicitată spre scriptul proxy, care o livrează apoi direct paginii client.

Mai întâi construiţi pagina client, similar cazului precedent dar identificatorul P1 e trimis prin metoda POST.

```
<html>
<head>
<script>
function solicitare()
		{
		cerere=new XMLHttpRequest()
		cerere.onreadystatechange=procesareRaspuns
		cerere.open("POST","scriptproxy3.php")
		cerere.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")
		cerere.send("identificator=P1")
		}
function procesareRaspuns()
		{
		if (cerere.readyState==4)
				if (cerere.status==200)
				{
				tinta=document.getElementById("tinta")
				tinta.innerHTML+=cerere.responseText
				}
		}
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

Observaţi modificările:

- datele trimise prin POST se trec în send, nu mai apar lipite la adresa destinaţie
- în open se declară metoda POST
- trebuie setat antetul HTTP cu setRequestHeader, astfel încât să se indice că trimitem un pachet de tip **application/x-www-form-urlencoded** (acesta e formatul standard pentru date culese din

formulare, deoarece POST se foloseşte de regulă pentru seturi mai mari de date culese dintr-un formular – nu şi aici, pentru că am păstrat exemplul minimal).

Mai departe, modificăm scriptul proxy (scriptproxy3 în site1):

```
<?php
$cerere=curl_init();
$configurari=array(    CURLOPT_RETURNTRANSFER=>1,
                       CURLOPT_URL=>"http://site2.com/datejsonpost.php",
                       CURLOPT_POST=>1,
                       CURLOPT_POSTFIELDS=>"identificator=".$_POST["identificator"]);
curl_setopt_array($cerere,$configurari);
$rezultat=curl_exec($cerere);
print $rezultat;
curl_close($cerere);
?>
```

Observaţi configurările cererii, unde am activat metoda POST (cu CURLOPT_POST) şi am specificat datele trimise (cu CURLOPT_POSTFIELDS ce preia valoarea primită de la client tot prin $_POST). Observaţi şi că procesarea datelor a dispărut de aici, scriptul proxy fiind redus la rolul de a forwarda datele în ambele sensuri.

În sfârşit, modificaţi scriptul remote (datejsonpost.php, în site2):

```
<?php
$datejson='{"Comanda":{"Client":"Poplon","Produse":[{"ID":"P1","Pret":100,"Denumire":"Televizor"},{"ID":"P2","Pret":30,"Denumire":"Ipod"}]}}';
$datephp=json_decode($datejson);
$identificator=$_POST["identificator"];
$produse=$datephp->Comanda->Produse;
foreach ($produse as $produs)
        {
        if ($produs->ID==$identificator) {$denumire=$produs->Denumire; break;}
        }
print $denumire;
?>
```

De data aceasta presupunem că pachetul JSON e deja disponibil într-un string, îl decodăm şi extragem din el denumirea pe baza identificatorului primit prin POST.
Reflectaţi asupra structurii JSON folosite. Observaţi că e nevoie de acel ciclu foreach pentru a găsi mai întâi produsul pe bază de ID şi apoi a extrage denumirea. Putem evita acest ciclu foreach dacă avem o structură JSON mai eficientă, ca în următorul exemplu:

```
<?php
$datejson='{"Comanda":{"Client":"Poplon","Produse":{"P1":{"Pret":100,"Denumire":"Televizor"},"P2":{"Pret":30,"Denumire":"Ipod"}}}}';
$datephp=json_decode($datejson,true);
$identificator=$_POST["identificator"];
$denumire=$datephp["Comanda"]["Produse"][$identificator]["Denumire"];
print $denumire;
?>
```

În noua structură JSON atributul ID a dispărut din produse, valorile sale devenind chei ce etichetează direct toate proprietăţile unui produs. Consecinţe:

- la decodare folosim argumentul suplimentar true pentru a putea folosi în PHP sintaxa vectorială (cu paranteze) în locul celei obiectuale (cu săgeţi); amintiţi-vă că în JavaScript ambele sintaxe pot fi folosite implicit, însă în PHP avem de ales una din variante;
- apoi accesăm denumirea prin această sintaxă vectorială, ce ne permite să folosim cheia P1 preluată din stringul $identificator; putem folosi şi sintaxa obiectuală, dar atunci suntem nevoiţi să asigurăm conversia stringului "P1" în proprietatea obiectuală P1, cu construcţia `$datephp->Comanda->Produse->{$identificator}->Denumire`

În sfârşit, reţineţi şi posibilitatea de a TESTA cereri HTTP cu ajutorul unor programe precum Postman[22]. Se obţine gratuit de la https://www.getpostman.com/ şi permite să executăm cereri HTTP într-o interfaţă grafică pentru a testa cum răspunde un script anume la o cerere formulată într-un anumit mod, cu anumite date.

Acest lucru e important din cel puţin două puncte de vedere:

- Ca instrument de testare în general;
- Atunci când lucrăm cu mai multe scripturi înlănţuite, cum e cazul acestui exemplu, poate dorim să testăm un singur script din lanţ, înainte să le construim pe celelalte. De exemplu dacă începem cu scriptul remote datejson.php, testarea lui presupune să-i trimitem identificatorul printr-o cererere de tip POST. Dacă ar fi o cerere GET, am putea-o testa direct în browser, lipind parametrul la adresă. Însă pentru metoda POST (sau alte metode HTTP) trebuie să folosim astfel de programe pentru a face toate configurările cererii. În imaginea de mai jos am construit o cerere POST spre adresa site2.com/datejson.php, iar în secţiunea Body am indicat parametrul **identificator** cu valoarea **P1**. Apăsarea butonului Send execută cererea, iar dedesubt se vede răspunsul.

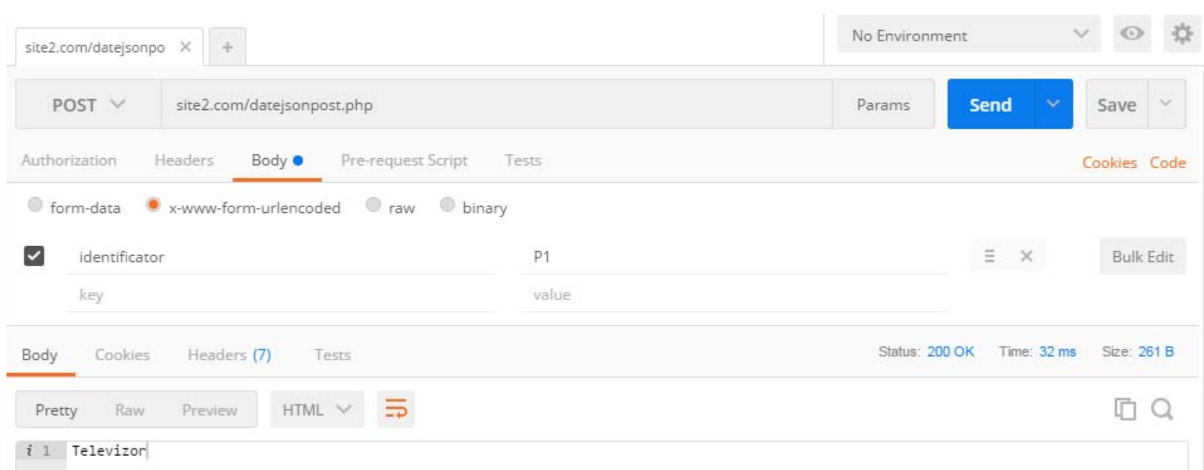


Figura 25 Trimitere cereri HTTP prin Postman

Postman e un program desktop bazat pe cURL[23], dar puteţi folosi şi diverse add-on-uri de browser în acelaşi scop.

2.5 Crearea de cereri HTTP cross-domain prin tehnica CORS

A treia tehnică ce poate fi folosită pentru cereri cross-domain din JavaScript e tehnica CORS[24], ce îmbină avantajele celorlalte două tehnici (avantajul JSON-P de a oferi acces neintermediat la surse remote şi avantajul scripturilor proxy de a permite mai mult decât cereri GET). Este, aşadar, metoda optimă însă a devenit posibilă doar recent, nu e suportată în toate versiunile de browsere şi presupune ca proprietarii serverului remote să realizeze anumite configurări. Acestea pot fi realizate în două moduri: (1) la nivel de server, dacă dorim ca toate scripturile executate pe acel server să se conformeze aceloraşi reguli; (2) la nivel de script.

Pentru configurări la nivel de server puteţi consulta detalii privind mai multe tipuri de servere la adresa https://enable-cors.org/server.html. Vom exemplifica în continuare doar cu configurări la nivel de script.
În forma cea mai simplă, o cerere CORS nu diferă cu nimic de o cerere XMLHttpRequest obişnuită, deci pagina client poate arăta la fel ca în alte situaţii discutate deja (salvaţi-o în site1):

[22] https://www.getpostman.com/
[23] https://github.com/curl/curl
[24] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

```
<html>
<head>
<script>
function solicitare()
        {
        cerere=new XMLHttpRequest()
        cerere.onreadystatechange=procesareRaspuns
        cerere.open("GET","http://site2.com/scriptcors.php")
        cerere.send(null)
        }
function procesareRaspuns()
        {
        if (cerere.readyState==4)
                if (cerere.status==200)
                {
                raspuns=JSON.parse(cerere.responseText)
                produs=raspuns.Comanda.Produse[0].Denumire
                tinta=document.getElementById("tinta")
                tinta.innerHTML+=produs
                }
        }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

În scriptul server remote (în site 2, cu numele scriptcors.php) se adaugă o singură linie la început, ce setează antetul HTTP Access-Control-Allow-Origin:

```
<?php
header("Access-Control-Allow-Origin: *");
$Produs1=array("ID"=>"P1","Pret"=>100,"Denumire"=>"Televizor");
$Produs2=array("ID"=>"P2","Pret"=>30,"Denumire"=>"Ipod");
$Produse=array($Produs1,$Produs2);
$Raspuns=array("Comanda"=>array("Client"=>"PopIon","Produse"=>$Produse));
print json_encode($Raspuns);
?>
```

Dacă valoarea setată este *, înseamnă că scriptul server acceptă cereri de la orice pagină client, din orice domeniu! Alternativ, proprietarul serverului poate restricţiona valoarea la unul sau mai multe domenii "client" explicit indicate, eventual preluate dintr-o bază de date proprie cu "clienţi agreaţi" (scriptul ar putea fi un serviciu public pe bază de abonare, în care dezvoltatorii de aplicaţii client ar putea să se înscrie, domeniile lor fiind apoi concatenate în acest câmp de antet HTTP pentru a activa permisiunea).

În practică însă ambele componente devin ceva mai complicate:

- Cererile CORS nu sunt suportate de orice browser, ci doar de cele care folosesc o versiune recentă a clasei XMLHttpRequest (ce se poate testa verificând existenţa proprietăţii nou-introduse withCredentials) sau cele care dispun de clasa XDomainRequest (cazul Internet Explorer);
- Există două tipuri de cereri CORS: **cereri simple** şi **cereri de tip preflighted.** În a doua categorie intră:
    - o cereri bazate pe alte metode decât GET şi POST
    - o cereri POST prin care clientul trimite alte structuri de date decât obişnuitele perechi nume=valoare (deci trimite alt Content-Type decât cel specific datelor colectate din formulare, al cărui cod MIME e application/x-www-form-urlencoded).

Primul aspect este tratat printr-o procedură mai complicată de instanţiere a cererii, care testează cu un IF diverse combinaţii:

```
function cerereCORS(metoda, adresa)
	{
	cerere=new XMLHttpRequest()
	if ("withCredentials" in cerere) cerere.open(metoda,adresa,true)
	else if (typeof XDomainRequest != "undefined")
				{
				cerere=new XDomainRequest()
				cerere.open(metoda,adresa)
				}
		else {cerere=null}
	return cerere
	}
function solicitare()
	{
	cerere=cerereCORS("GET","http://site2.com/scriptcors.php")
	if (cerere)
		{
		cerere.onload=procesareRaspuns
		cerere.send()
		}
	else alert("browserul nu suporta CORS")
	}
function procesareRaspuns()
	{
	raspuns=JSON.parse(cerere.responseText)
	produs=raspuns["Comanda"]["Produse"][0]["Denumire"]
	tinta=document.getElementById("tinta")
	tinta.innerHTML+=produs
	}
```

Observaţii:

- Mai întâi se testează existenţa proprietăţii withCredentials, prezentă doar în versiuni mai noi ale clasei XMLHttpRequest (în unele browsere recente). Apoi se testează existenţa clasei XDomainRequest (folosită de Internet Explorer). Dacă niciuna din condiţii nu e îndeplinită, înseamnă că browserul nu suportă tehnica CORS;
- În funcţia solicitare(), în caz că una din aceste instanţieri a avut succes, se alocă funcţia de procesare a răspunsului cu **onload** – această funcţie apare în loc de onreadystatechange la browsere mai recente (aduce beneficiul că nu mai trebuie incluse testele de status şi readyState pentru a verifica succesul sosirii răspunsului).

Totuşi, pentru a simplifica exemplele în continuare vom utiliza doar browserul Chrome în versiunea cea mai recentă şi vom folosi clasa XMLHttpRequest fără celelalte verificări.

În următorul exemplu vom realiza o **cerere preflighted**. În această categorie intră:

- acele cereri care nu folosesc metodele GET şi POST (ci PUT, DELETE etc., frecvent folosite pentru a realiza operaţii pe baze de date on-line fără a apela la interogări SQL); vom reveni ulterior asupra acestui subiect;
- cererile POST care trimit spre server structuri mai complexe decât simple perechi **nume=valoare**, de exemplu cod XML sau JSON! (cu cererile simple exemplificate până acum am trimis spre server doar parametri simpli de forma nume=valoare, sau n-am trimis nimic ci doar am solicitat ceva de la server).

Salvaţi pagina client în site1:

```
<html>
<head>
<script>
function solicitare()
	{
	studenti=[{"Nume":"Ana","Nota":10},{"Nume":"Petru","Nota":4}]
	datejson=JSON.stringify(studenti)
```

```
            cerere=new XMLHttpRequest()
            cerere.open("POST","http://site2.com/script2cors.php")
            cerere.setRequestHeader("Content-Type","application/json")
            cerere.onload=procesareRaspuns
            cerere.send(datejson)
            }
function procesareRaspuns()
            {
            tinta=document.getElementById("tinta")
            tinta.innerHTML+=cerere.responseText
            }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

Observaţii:

- JSON.stringify e operaţia inversă lui JSON.parse. Practic am convertit un vector de obiecte JavaScript într-un string JSON pe care îl trimitem serverului;
- Am construit o cerere POST spre serverul site2. Deoarece trimitem cod JSON şi nu nişte simpli parametri trebuie să anunţăm serverul prin setarea antetului HTTP "Content-Type".

Pe server (site2), scriptul care primeşte o cerere preflight trebuie să realizeze minim următoarele operaţii (script2cors.php):

```
<?php
header("Access-Control-Allow-Origin: *");

if ($_SERVER["REQUEST_METHOD"]=="OPTIONS")
            {
            header("Access-Control-Allow-Methods: OPTIONS,POST");
            header("Access-Control-Allow-Headers: Content-Type");
            header('Access-Control-Max-Age: 1');
            }
if ($_SERVER["REQUEST_METHOD"]=="POST")
                        {
                        if (($_SERVER["HTTP_ORIGIN"]=="http://site1.com")&&($_SERVER["CONTENT_TYPE"]=="application/json"))
                                    {
                                    $dateClient=file_get_contents("php://input");
                                    $datePHP=json_decode($dateClient);
                                    print "Serverul confirma primirea de note pentru ".$datePHP[0]->Nume." si ".$datePHP[1]->Nume;
                                    }
                        else {print "Acceptam cereri POST doar de la site1.com si doar cu continut JSON";}
                        }
else print "Nu acceptam cereri GET!";
?>
```

O cerere preflighted este de fapt o succesiune de 2 cereri: o **pre-cerere** de verificare generată automat de browser (prin metoda HTTP OPTIONS) apoi cererea efectivă prin metoda specificată de client (aici POST).

1. Prin pre-cerere browserul verifică dacă serverul suportă tehnica CORS şi îl întreabă ce metode şi antete HTTP acceptă. Această pre-cerere este de tip OPTIONS, deci scriptul server trebuie să conţină un IF care să detecteze pre-cererea şi să-i răspundă cu informaţiile relevante. De regulă această informaţie "relevantă" e o succesiune de câmpuri de antet HTTP prin care clientul e informat asupra a ce are voie să ceară:
    - antetul *Access-Control-Allow-Methods* indică browserului ce metode HTTP acceptă serverul (lista trebuie să includă metoda OPTIONS necesară pre-cererii, la care se adaugă una sau mai multe metode suportate de server; în exemplul de faţă serverul mai suportă doar cereri prin metoda POST – ceea ce e suficient pentru nevoile paginii client);

- antetul *Access-Control-Allow-Headers* indică browserului ce antete HTTP acceptă serverul (aici "Content-Type", prin care clientul poate anunţa faptul că trimite date în alte formate decât simpli parametri nume=valoare);
- antetul *Access-Control-Max-Age* stabileşte cât timp să se stocheze în browser (cache) aceste informaţii; în timpul dezvoltării recomandăm să fie 1 secundă, pentru a putea vedea efectul eventualelor corecturi, însă la lansarea site-ului se pun câteva ore, astfel încât pre-cererea să nu mai fie repetată la fiecare cerere CORS (având în vedere că aceste informaţii nu se prea modifică);

2. După ce pre-cererea a fost finalizată şi există o confirmare că serverul e pregătit să accepte ce are clientul de trimis, browserul va trimite cererea efectivă. În exemplul de faţă, clientul e compatibil cu condiţiile impuse de server, deoarece va trimite ceva prin metoda POST, într-un format anunţat cu antetul Content-Type. Scriptul server trebuie să fie pregătit, într-o altă ramură IF, să accepte şi cererea efectivă:
   - remarcaţi IF-ul care verifică faptul că a sosit o cerere POST, apoi cel care verifică dacă sursa cererii e *site1* iar tipul datelor e application/json (prin acest IF scriptul limitează sursa la site1, chiar dacă pe prima linie restricţia e relaxată; prin asta încercăm să sugerăm că serverul poate pregăti mai multe ramuri IF care să trateze clienţi diferiţi în moduri diferite);
   - datele sosite prin POST sunt preluate din file_get_contents("php://input") şi nu din $_POST cum suntem obişnuiţi (asta deoarece nu mai e vorba de variabile simple, ci de o structură complexă de date);
   - am construit un mesaj care confirmă că datele au sosit şi acest mesaj e returnat clientului pentru afişare.
3. Se pot adăuga şi alte IF-uri în mod similar, care să ofere reacţii diferite la cereri de alte tipuri (GET, PUT, DELETE etc.). Aici nu mai avem altele, aşa că am adăugat o ramură ELSE anunţând printr-un mesaj că cereri GET nu se acceptă.

Testaţi exemplul în următoarele moduri:

- Executaţi pagina client (în consola browserului puteţi observa că au loc 2 cereri una după alta);
- Accesaţi direct scriptul server prin browser – această tentativă va accesa scriptul prin metoda GET, ceea ce va duce la mesajul de eroare de la finalul scriptului;
- Accesaţi direct scriptul server prin Postman, configurând o cerere POST ca în imagine (cu datele incluse în secţiunea Body, tipul raw). Veţi primi penultimul mesaj de eroare, declanşat de faptul că originea cererii nu e site1.com!

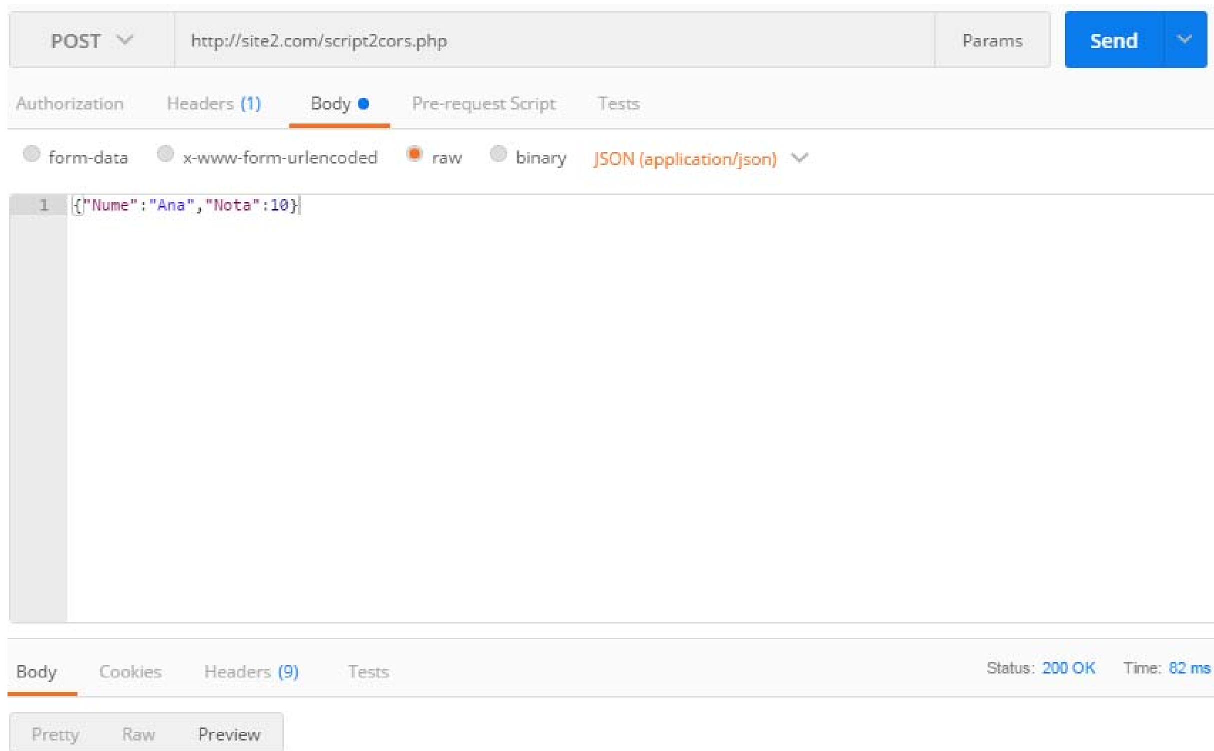


Figura 26 Trimitere cerere POST către scriptul server de pe site2.com prin Postman

În sfârşit, în următorul exemplu realizăm cererea CORS şi prin Jquery:

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function solicitare()
        {
        studenti=[{"Nume":"Ana","Nota":10},{"Nume":"Petru","Nota":4}]
        datejson=JSON.stringify(studenti)
        configurari={url:"http://site2.com/script2cors.php",type:"POST",data:datejson,contentType:"application/json",success:procesareRaspuns}
        $.ajax(configurari)
        }
function procesareRaspuns(raspuns)
        {
        $("#tinta").html(raspuns)
        }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

Observaţie:

- Suntem nevoiţi să folosim $.ajax, deşi Jquery oferă şi varianta mai scurtă $.post pentru cereri POST. Motivul este că $.post nu permite modificarea antetului Content-Type (situaţia se poate schimba în versiuni viitoare de Jquery).
- Observaţi şi noul mod prin care am inserat răspunsul în pagină, de data aceasta realizând o selecţie după ID şi folosind metoda html() (comparaţi cu modul folosit anterior, cu appendTo).

2.6 Crearea de cereri spre servere JSON

Până la acest punct am construit răspunsuri JSON cu ajutorul unui script PHP. Sursa acelor date ar putea fi bazele de date MySQL, din care PHP să le preia şi să le împacheteze în JSON. Sau, după cum am văzut la tehnica scripturilor proxy, sursa ar putea fi alte scripturi de pe alte servere.

O situaţie tot mai frecventă în ultimii ani este ca datele JSON să fie solicitate direct de la **servicii de date JSON** care:

- Pot fi percepute ca nişte baze de date în care toată informaţia e stocată în format JSON;
- Accesarea datelor se realizează on-line prin interogări "deghizate" în cereri HTTP (de aceea, în asociere cu aceste tehnologii veţi mai întâlni termeni precum "baze de date NoSQL" sau "Database-as-a-service"); mai exact, clienţii nu vor interoga direct baza de date, ci interacţionează cu ea printr-o interfaţă ce permite cereri HTTP cu diverse combinaţii de configurări.

Există o serie de servicii JSON disponibile on-line, cu diverse limitări (de preţ, de număr de cereri etc.) însă pentru a fi cât mai flexibili, vom instala propriul server JSON gratuit, care va putea fi "interogat" prin cereri HTTP trimise fie din JavaScript, fie din PHP (şi din orice alt limbaj ce oferă posibilitatea de a executa cereri HTTP).

Serverul pentru care optăm pentru exemplificare e **JSON Server**[25] şi trebuie instalat peste frameworkul **Node.js** (un framework popular pentru lucrul server-side cu JavaScript). Paşii instalării şi pornirii sunt:

Pas1. Descărcaţi şi instalaţi NodeJS de la adresa de mai jos, asigurându-vă că în timpul instalării sunt bifate toate componentele, inclusiv componenta NPM (aceasta se ocupă de descărcarea ulterioară a altor pachete/librării bazate pe Node.js, printre ele numărându-se şi JSON Server).
https://nodejs.org/en/

Pas2. Executaţi în linia de comandă Windows comanda de mai jos, ce se ocupă de descărcarea şi instalarea JSON Server:

```
npm install –g json-server
```

Pas3. Creaţi un fişier JSON care să conţină datele iniţiale ce le vom încărca în baza de date. Salvaţi acest fişier cu numele dateinitiale.json (în htdocs) şi cu conţinutul următor:

```
{
"students": [
        {"id": 1,
        "name": "Pop Ion"},
        {"id": 2,
        "name": "Pop Maria"}
        ],
"courses": [
        {"id": 1,
        "title": "Web development",
        "teacher":{"name":"Popescu Ion","office":404}},
        {"id": 2,
        "title": "Java",
        "teacher":{"name":"Ionescu Maria","office":403}},
        {"id": 3,
        "title": "Databases",
        "teacher":{"name":"Marian Vasile","office":401}}
        ],
"grades": [
        {"courseId":1,
        "studentId":1,
        "grade":7},
        {"courseId":1,
```

[25] Detalii complete despre JSON Server puteţi găsi la https://github.com/typicode/json-server

```
"studentId":2,
"grade":5},
{"courseId":2,
"studentId":1,
"grade":5},
{"courseId":2,
"studentId":2,
"grade":5},
{"courseId":3,
"studentId":2,
"grade":10}
],
"faculty":{"name":"FSEGA","address":"str. T Mihali 58-60 Cluj Napoca"}
}
```


Pas4. Navigaţi cu linia de comandă Windows până în folderul unde aţi salvat fişierul:

```
cd c:\xampp\htdocs
```

Pas5. Fiind poziţionaţi în folderul cu fişierul, porniţi JSON server precizând totodată să stocheze datele din fişierul JSON creat:

```
json-server --watch dateinitiale.json --port 4000
```

Opţiunea --port stabileşte portul localhost la care se pot trimite cererile spre JSON server.
Opţiunea --watch face ca orice modificare în fişierul cu date iniţiale să se transmită automat la datele serverului. Asta înseamnă că modificările salvate în fişierul dateinitiale.json se vor sincroniza automat cu cele din server şi vor deveni imediat accesibile pentru "interogări".

Accesaţi în browser localhost:4000 şi veţi vedea pagina de întâmpinare JSON Server.

Analizaţi puţin fişierul JSON, făcând o paralelă cu bazele de date MySQL. Ceea ce avem în fişier este practic echivalentul a trei tabele:

- students (cu câmpurile id şi name),
- courses (cu câmpurile id, title şi teacher, ultimul având valori complexe de tip obiect)
- grades (cu câmpurile studentId - care funcţionează ca o "cheie străină", indicând ID-ul uneia din înregistrările din students; courseId – la fel, cheie străină ce stabileşte o relaţie cu cursurile; grade – nota pe care studentul referit a luat-o la cursul referit); practic prin "tabelul" grades am realizat o relaţie many-to-many între studenţi şi cursuri;
- la acestea se mai adaugă vectorul asociativ faculty, care nu e considerat neapărat tabel, cât mai degrabă ca un obiect unic cu o serie de proprietăţi (name, address)

Reţineţi:

- putem considera fiecare vector de pe primul nivel ierarhic al fişierului JSON ca fiind un "tabel" în care cheia primară obligatoriu trebuie să poarte numele "id" (cu valori unice!) iar cheile străine poartă un nume format din singularul numelui tabelului concatenat cu Id: dacă dorim o relaţie cu "tabelul" students numele cheii străine trebuie să fie studentId (din acest motiv numele tabelelor trebuie să fie în engleză)
- pe lângă tabele, fişierul JSON poate conţine pe primul nivel ierarhic şi obiecte singulare ce nu pot fi interpretate ca "tabele" (vezi faculty).

Toate entităţile descoperite pe primul nivel ierarhic al fişierul JSON (adică cele 3 tabele şi obiectul faculty) sunt considerate "resurse" şi vor putea fi accesate direct prin cereri HTTP. Puteţi trimite cereri de tip GET direct în bara de adresă a browserului, însă pentru alte tipuri de cereri folosiţi fie programul Postman, fie scripturi.

Începem cu câteva cereri GET tastate direct în browser, care practic funcţionează ca nişte interogări.

http://localhost:4000/db
(afişează tot conţinutul bazei de date)

http://localhost:4000/students
(afişează datele tuturor studenţilor)

http://localhost:4000/students?_sort=name&_order=DESC
(afişează datele tuturor studenţilor sortate descrescători după nume)

http://localhost:4000/faculty
(afişează datele obiectului faculty)

http://localhost:4000/students/1
(afişează datele studentului cu id=1)

http://localhost:4000/students?name=Pop Ion (spaţiul se va substitui automat)
(afişează datele studentului cu name="Pop Ion"; se pot adăuga mai multe filtre similare separate cu &)

http://localhost:4000/students?id_ne=2
(afişează datele studentului cu id diferit de 2; codul **_ne=** se interpretează ca **not equal**)

http://localhost:4000/students?q=Maria
(afişează datele studentului în ale cărui date apare stringul "Maria")

http://localhost:4000/grades?grade_gte=6&grade_lte=9
(afişează notele cu grade>=6 şi <=9; observaţi codurile folosite: **_gte** şi **_lte**, pentru că nu putem folosi caracterele > sau < în adrese URL)

Observaţie: JSON Server creează "tabele" doar din entităţile pe care le găseşte pe primul nivel ierarhic. Entităţile de pe nivele ierarhice mai adânci (cum este teacher) NU vor putea fi accesate direct – putem obţine datele lor laolaltă cu restul entităţii din care fac parte (courses), dar nu putem formula o cerere directă care să ne dea doar profesorii, de genul http://localhost:4000/courses/1/teacher[26]. În schimb putem implica datele profesorilor în condiţii:

http://localhost:4000/courses?teacher.office=403
(afişează datele cursului ale cărui profesor are biroul 403; observaţi cum informaţiile despre profesori pot fi accesate cu ajutorul punctului, însă numai în scopul testării de condiţii)

Observaţie: Am amintit anterior că prin cheile străine studentId şi courseId avem practic o relaţie între "tabele". Putem beneficia de relaţia respectivă în următoarele moduri:

http://localhost:4000/grades?_expand=student&_expand=course
(afişează notele, expandând informaţia despre studenţi şi cursuri preluată din tabelele relaţionate).

http://localhost:4000/students/1?_embed=grades
(afişează datele studentului cu ID=1 incluzând notele acestora, detectate tot pe baza corespondenţei students-studentId; la fel putem obţine şi cursuri cu notele incluse)

http://localhost:4000/students/1/grades
(afişează notele studentului 1, beneficiind de relaţia grades-students)

http://localhost:4000/students/1/grades?courseId=2
(afişează nota studentului 1, la cursul 2)

[26] Dacă dorim să facem posibile astfel de interogări trebuie să mutăm profesorii pe primul nivel al fişierului şi să construim relaţii cu teacherId, aşa cum am făcut şi cu studentId şi courseId. Sintaxa JSON ne permite înglobarea multiplă de obiecte unele în altele, însă JSON Server va accepta interogări directe doar pentru cele de pe primul nivel ierarhic.

Observaţie: Toate aceste "interogări" returnează vectori sau obiecte COMPLETE în format JSON. Nu putem extrage o valoare simplă (de genul "titlul cursului 1"). Valorile simple vor trebui extrase de clientul care trimite cererea HTTP (în acele funcţii de procesare a răspunsului care apar în exemplele JavaScript).

Toate exemplele de până acum au fost cereri GET tastate direct în browser. Pentru alte tipuri de cereri folosiţi programul Postman. Se mai acceptă următoarele tipuri:

- POST pentru upload de date în server (similar comenzii INSERT din SQL);
- PATCH pentru editarea proprietăţilor unei entităţi (similar comenzii UPDATE din SQL)
- PUT pentru substituirea de "înregistrări" (similar unui UPDATE care modifică toate câmpurile unei înregistrări în afara cheii primare; atenţie, operaţia poate să şi şteargă câmpuri cu totul)
- DELETE pentru ştergere de "înregistrări"

Mai întâi construiţi în Postman o cerere POST cu elementele:

- Adresa http://localhost:4000/students
- Metoda POST
- Headers: Content-Type=application/json (acesta trebuie setat la ORICARE din cererile care trimit date, şi datele trebuie să fie structuri JSON complete, nu se pot trimite valori simple)
- Body: raw JSON cu valoarea {"name":"Ionescu Andrei"}

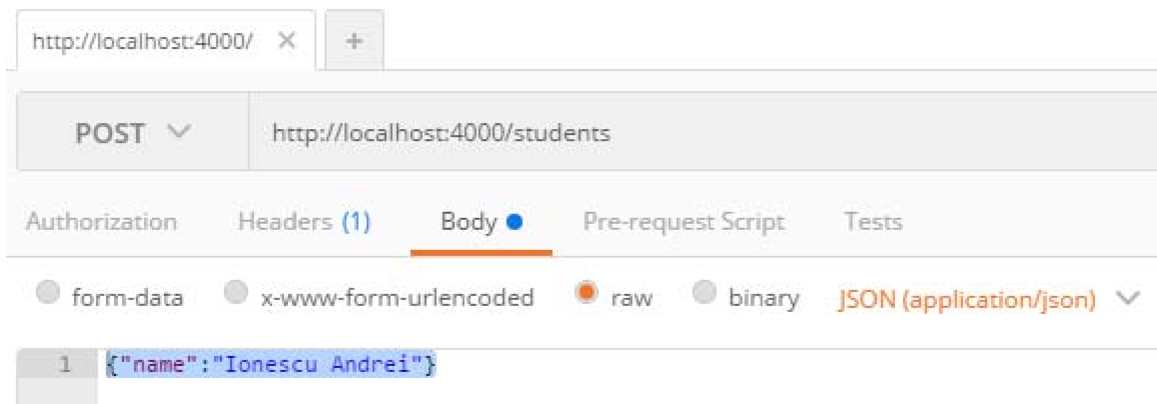


Figura 27 Cerere POST către serverul JSON prin Postman

Efectul este inserarea unui student nou în vectorul students, cu numele Ionescu Andrei. ID-ul nu trebuie precizat deoarece cheia primară e incrementată automat de JSON-Server (avem voie să precizăm un ID, dar riscăm să introducem valori duplicate aşa că cel mai comod e să lăsăm JSON Server să gestioneze ID-urile).

Pentru a vedea efectul închideţi şi redeschideţi fişierul dateinitiale.json, care e menţinut permanent sincron cu serverul! (dacă serverul a fost pornit cu --watch). Dacă aveţi deja fişierul deschis în Notepad++ acesta va avertiza că fişierul s-a modificat, dacă îl aveţi deschis în VS Code probabil se va actualiza automat.

Construiţi şi o cerere DELETE pentru a şterge studentul adăugat:

- Adresa http://localhost:4000/students/3 (remarcaţi ID-ul studetului, acesta a fost generat de POST-ul anterior!)
- Metoda DELETE
- Headers: Content-Type nu contează pentru că nu se trimit date
- Body rămâne gol deoarece nu se trimit date

Construiţi şi o cerere PATCH pentru a modifica datele profesorului de la cursul cu ID=2:

- Adresa http://localhost:4000/courses/2
- Metoda PATCH
- Headers:Content-Type=application/json
- Body cu datele noi ale profesorului: {"teacher": {"name": "Moldovan Ioan","office": 409}}

Construiţi şi o cerere PUT pentru a înlocui toate detaliile facultăţii – nu doar valorile proprietăţilor, ci chiar proprietăţile:

- Adresa http://localhost:4000/faculty
- Metoda PUT
- Headers:Content-Type=application/json
- Body cu datele noi ale facultăţii: {"country":"Romania", "domain":"Economics"}

Construiţi şi o cerere PUT pentru a înlocui toate datele primului student, cu excepţia ID-ului (acesta nu poate fi modificat nici cu PUT, nici cu PATCH, aşa că nu trebuie inclus în datele trimise):

- Adresa http://localhost:4000/students/1
- Metoda PUT
- Headers:Content-Type=application/json
- Body cu datele noi {"nume de familie":"Popovici","prenume":"Andrei"}

Observaţi că nu e doar o modificare de valori, ci o substituire completă – practic am introdus proprietăţi noi care nu mai respectă structura generală a entităţilor-studenţi. Bazele de date de tip JSON Server nu au o structură fixă asemenea bazelor de date relaţionale – noi obiecte şi proprietăţi pot fi adăugate, şterse, create oricând. Aceasta poate fi percepută pe de o parte ca flexibilitate, pe de altă parte ca un pericol de a produce inconsistenţe şi date invalide. Serverele comerciale oferă mecanisme de a controla validitatea şi structura - aşa cum documentele XML pot fi verificate cu ajutorul XML Schema, şi structurile JSON pot fi verificate cu ajutorul JSON Schema.

Modificaţi numele de familie al studentului introdus cu ajutorul unei cereri PATCH:

- Adresa http://localhost:4000/students/1
- Metoda PATCH
- Headers:Content-Type=application/json
- Body cu datele noi {"nume de familie":"Popescu"}

Observaţi că în metoda PATCH se modifică doar valorile trimise, restul se păstrează, în timp ce cu metoda PUT se înlocuiesc complet proprietăţile cu cele pe care le trimitem (şi valorile trimise), deci proprietăţile netrimise prin PUT dispar, în timp ce proprietăţile netrimise prin PATCH se păstrează.

În final realizăm o operaţie PUT dintr-o pagină client, prin Jquery:

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function solicitare()
        {
        proprietati={"first name":"Maria","last name":"Pop"}
        datejson=JSON.stringify(proprietati)
        configurari={url:"http://localhost:4000/students/2",type:"PUT",data:datejson,contentType:"application/json",
                                                                        success:procesareRaspuns}
        $.ajax(configurari)
        }
function procesareRaspuns(raspuns)
        {
        $("#tinta").html("s-a inserat cu succes studentul "+raspuns["first name"]+" "+raspuns["last name"])
        }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

Observaţi modul de configurare a cererii ce respectă indicaţiile anterioare (ţintirea studentului 2 prin adresa URL, metoda PUT pentru înlocuirea datelor, declararea contentType). JSON Server răspunde implicit cu informaţia care s-a adăugat, deci din variabila raspuns putem accesa toate proprietăţile inserate şi le accesăm pentru a afişa un mesaj de confirmare.
La executarea scriptului verificaţi faptul că studentul 2 din dateinitiale.json i s-au înlocuit datele cu cele trimise din script. Succesul scriptului ne demonstrează şi faptul că JSON Server este deja configurat pentru cereri prin tehnica CORS.

Mai mult, JSON Server este configurat şi pentru tehnica JSON-P. Putem demonstra asta trimiţând o cerere GET cu parametrul callback, care să ne listeze toate cursurile din baza de date şi profesorii lor:

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function solicitare()
        {
        adresa="http://localhost:4000/courses?callback=?"
        $.getJSON(adresa,function(raspuns)
                        {
                        $.each(raspuns,function(indice,curs)
                                        {
                                        continutDeAfisat="<div><i>"+curs.title+"</i> predat de "+curs.teacher.name+"</div>"
                                        $(continutDeAfisat).appendTo(document.body)
                                        }
                            )
                        }
                )
        }
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<p>Cursurile din baza de date sunt:</p>
</body>
</html>
```

Observaţi:

- Adresa cererii, corespunzătoare unei interogări ce solicită toate cursurile, precum şi argumentul callback=? ce activează în Jquery tehnica JSON-P
- Metoda GET e implicită când folosim $.getJSON
- Funcţia anonimă de procesare a răspunsului foloseşte facilitatea $.each oferită de Jquery. Aceasta primeşte ca argument un vector (în acest caz vectorul cursurilor primit de la JSON Server) şi procesează fiecare element al vectorului printr-o funcţie anonimă
    - În cazul de faţă, din fiecare curs găsit în vector se construieşte un mesaj ce anunţă titlul şi profesorul identificat în datele cursului; fiecare mesaj e adăugat la pagină devenind vizibil în browser.

Ultimele exemple demonstrează şi posibilitatea de a crea pagini Web dinamice care nu mai au deloc un script PHP pe server. Rolul bazei de date MySQL e jucat de serviciul oferit de JSON Server, rolul interogărilor SQL este preluat de cererile HTTP iniţiate direct din pagina HTML cu ajutorul Jquery (sau XMLHttpRequest, sau alte posibilităţi oferite de JavaScript). Desigur, am putea avea şi un script PHP care să acceseze JSON Server şi să aplice prelucrări suplimentare asupra datelor înainte să le ofere paginii client – această tehnică devine utilă atunci când se interoghează date de la mai multe surse şi e necesar un efort suplimentar de combinare a lor înainte să fie prezentate, de exemplu dacă există totuşi şi o bază de date MySQL proprie ale cărei date trebuie combinate cumva cu cele de la servicii cross-domain precum acest JSON Server.

2.7 Cereri asincrone în scripturi proxy PHP

Am văzut deja cum realizăm cereri HTTP în PHP, cu ajutorul bibliotecii cURL. Există o diferenţă importantă faţă de cererile HTTP realizate în JavaScript (cu Jquery sau XMLHttpRequest). Cererile exemplificate din JavaScript sunt **asincrone** (de aici litera A din AJAX) în timp ce cererile cURL între scripturi PHP sunt **sincrone**.

Prin **cerere asincronă** înţelegem o cerere care, dacă răspunsul îi întârzie (de exemplu dacă se aşteaptă cantităţi mari de date), restul scriptului continuă să funcţioneze, nu rămâne blocat într-o stare de aşteptare până la sosirea răspunsului. Aceasta e o cerinţă esenţială în site-urile moderne, unde clickurile utilizatorului declanşează adesea schimburi de date cu serverul, iar acestea nu trebuie să întrerupă funcţionarea paginii (ci cel mult a porţiunii de pagină care are nevoie de acele date). În site-urile vechi clickurile care solicitau informaţii de la server (butoanele submit, linkurile) aveau ca efect reîncărcarea integrală a paginii, de aceea PHP genera pagini HTML întregi şi nu doar date.

În prezent cererile asincrone sunt un element obligatoriu în orice pagină client, realizate adesea cu Jquery şi alte metode. În schimb cererile cURL *între scripturi PHP* nu sunt asincrone – dacă trimitem mai multe cereri cURL din acelaşi script PHP, fiecare cerere va bloca temporar execuţia, până când soseşte răspunsul (cererile se vor executa în ordinea în care apar în script). În PHP cererile asincrone devin importante atunci când avem nevoie de *mai multe cereri spre mai multe surse diferite şi dorim ca acele cereri să se execute în paralel, fără a aştepta unele după altele*.

Pentru a realiza cereri asincrone în PHP nu mai lucrăm cu cURL, ci avem nevoie de una din librăriile care oferă această facilitate. Cea mai populară librărie este **GuzzleHttp**. Orice putem face în cURL e posibil şi în GuzzleHttp, dar ultima oferă şi un mod comod de a realiza cereri paralele asincrone spre multiple servere sau servicii.

GuzzleHttp nu este inclusă în instalarea implicită pe care XAMPP o oferă pentru PHP, aşa că trebuie să o instalăm. Pentru instalare avem nevoie de programul **Composer**, care se ocupă de descărcarea şi instalarea a diverse extensii PHP (practic Composer este pentru PHP ceea ce este NPM pentru Node.js).

Vom realiza aşadar mai întâi instalarea programului Composer, apoi îl vom folosi pentru a instala extensia GuzzleHttp (iar în viitor vom mai instala şi alte extensii):

Pas1. Căutaţi fişierul php.ini din XAMPP (la o instalare normală ar fi în C:\xampp\php). În acest fişier asiguraţi-vă că linia *extension=php_openssl.dll* nu e dezactivată prin comentariu (dacă e, ştergeţi semnul ; din faţa ei)

Pas2. Descărcaţi şi executaţi Composer-Setup.exe de la adresa de mai jos:
https://getcomposer.org/download/
(în timpul instalării veţi fi întrebaţi unde e instalat php, selectaţi calea la care a fost instalat de XAMPP)

Pas3. Folosiţi linia de comandă Windows pentru a naviga în site1:
```
cd c:\xampp\htdocs\site1
```

Pas4. Aflaţi fiind în folderul site1, unde vom crea exemplul, instalaţi librăria GuzzleHttp cu comanda:
```
composer require guzzlehttp/guzzle
```
(comanda ar trebui să înceapă să descarce fişierele Guzzle şi să le stocheze pe toate într-un folder numit vendor; de acolo vor trebui importate în scripturile PHP ce au nevoie de Guzzle)

Documentaţia oficială GuzzleHttp poate fi consultată la adresa:
http://docs.guzzlephp.org/en/latest/

Considerăm că avem în site2 acelaşi script pe care l-am mai folosit, cu numele datejson.php:

```
<?php
header("Content-type:application/json");
$Produs1=array("ID"=>"P1","Pret"=>100,"Denumire"=>"Televizor");
$Produs2=array("ID"=>"P2","Pret"=>30,"Denumire"=>"Ipod");
$Produse=array($Produs1,$Produs2);
$Raspuns=array("Comanda"=>array("Client"=>"PopIon","Produse"=>$Produse));
print json_encode($Raspuns);
?>
```

Creaţi în site 1 următorul script, care contactează scriptul de mai sus prin Guzzle:

```
<?php
require "vendor/autoload.php";
$client=new \GuzzleHttp\Client();
$cerere=$client->getAsync("http://site2.com/datejson.php");
$cerere->then("procesareRaspuns")->wait();

function procesareRaspuns($raspuns)
		{
		print $raspuns->getBody();
		}
?>
```

Observaţii:

- Cu require ne asigurăm că toate funcţiile şi clasele Guzzle vor fi importate automat când le menţionăm în cod, fără a fi necesare operaţii multiple de import/includere (trebuie să fim atenţi la calea fişierului autoload, să fie corectă);
- Pe linia următoare am instanţiat clasa Client() oferită de GuzzleHttp;
- Apoi am creat o cerere asincronă de tip GET cu getAsync(), spre scriptul ce oferă date JSON;
- Pe linia următoare funcţia then() indicăm funcţia responsabilă cu procesarea răspunsului; cererea efectivă se execută înlănţuind mai departe wait();
- La final am construit funcţia de procesare a răspunsului separată de restul codului (aşa cum procedam şi în JavaScript); aceasta realizează doar o afişare a răspunsului integral preluat cu getBody() (echivalentul lui responseText din JavaScript).

După cum spuneam, beneficiile reale ale cererilor asincrone în PHP ţin de executarea paralelă a mai multor cereri spre mai multe destinaţii. De aceea vom folosi şi o a doua destinaţie – baza de date JSON Server de la exerciţiile precedente. Asiguraţi-vă că serverul este pornit şi cu datele încărcate:

```
json-server --watch dateinitiale.json --port 4000
```

Mai jos aveţi un exemplu de script care solicită date atât de la datejson.php cât şi de la JSON Server şi nu face altceva decât să le afişeze:

```
<?php
require "vendor/autoload.php";
$client=new \GuzzleHttp\Client();
$cereri=[
		"cerere1"=>$client->getAsync("http://site2.com/datejson.php"),
		"cerere2"=>$client->getAsync("http://localhost:4000/students")
		];
$rezultate=\GuzzleHttp\Promise\unwrap($cereri);
print $rezultate["cerere1"]->getBody();
print $rezultate["cerere2"]->getBody();
?>
```

Observaţi cum:

- am construit un vector de cereri asincrone spre cele două destinaţii
- am folosit funcţia Promise\unwrap() pentru a le executa pe ambele în paralel şi a culege rezultatele ambelor într-un vector de rezultate
- apoi am afişat din acest vector toate datele sosite.

GuzzleHttp poate construi şi cereri sincrone precum cURL, de diferite tipuri (POST, PUT, PATCH etc.), cu o sintaxă destul de comodă. Documentaţia oficială GuzzleHttp, unde găsiţi numeroase exemple de cereri Guzzle, poate fi consultată la adresa:

- http://docs.guzzlephp.org/en/latest/quickstart.html

3. Reprezentarea cunoștințelor în grafuri RDF

RDF[27] este un set de specificații pentru reprezentarea de "cunoștințe" (descrieri, afirmații) **sub formă de grafuri**. Acest lucru este diferit de XML, care este axat pe reprezentarea informaţiei **sub forma unui arbore** (DOM). Grafurile sunt mai flexibile decât arborii (care sunt mai flexibili decât tabelele) și se pot interoga în moduri mai puternice (într-un arbore se poate naviga de-a lungul relațiilor părinte-copil, în grafuri se poate naviga în orice direcție).

O bază de grafuri RDF poate fi considerată:

- O "bază de date NoSQL", deoarece este interogată cu limbajul SPARQL[28] în loc de SQL;
- O "bază de cunoștințe", deoarece este folosită de obicei pentru stocarea de cunoștințe (=afirmații, fapte, reguli) nu doar pentru stocarea de date (=valori de diferite tipuri),
- O "bază de date semantică" (deoarece este totuşi o bază de date, însă una în care datele sunt însoțite de semnificația acestora într-un mod în care "semnificația" respectivă poate fi interogată);
- O bază de date de tip "Linked Data" sau "Smart Data" (termeni folosiți mai mult în context de marketing pentru a populariza astfel de tehnologii)

3.1 Comparaţii cu XML

- La fel ca XML, RDF nu oferă un limbaj, ci un set de reguli de "bună formare" care trebuie să fie respectate în următoarele scopuri:
    - pentru a **descrie cunoștințe** sub forma unor **afirmaţii interogabile**
    - şi apoi pentru a conecta acele afirmaţii în **grafuri**.

  Fiecare afirmaţie conține 3 **termeni**: un subiect, un predicat (proprietate) și un obiect.
- Aşa cum în XML putem folosi orice marcatori sau atribute dorim, şi în grafuri RDF putem folosi orice "termeni" - putem inventa proprii noștri termeni, putem folosi termeni creaţi de altcineva, putem adopta termeni standardizaţi sau orice combinaţie între aceştia.
- La fel ca în XML, dacă dorim ca alții să înțeleagă afirmaţiile noastre, este indicat să adoptăm o **terminologie standard**, adică un set de termeni standardizaţi recunoscuţi de numeroase instrumente software și de programatori din toată lumea care ar putea să interogheze grafurile noastre. Cele mai importante terminologii standard sunt Schema.org, RDF/S[29] și OWL[30].
- La fel ca în XML, putem construi vocabulare (aici numite şi **ontologii**, **terminologii, taxonomii**).
    - *Un vocabular RDF se aseamănă cu un vocabular XML* prin aceea că stabileşte un acord între mai multe organizaţii privind termenii care să fie folosiţi (aici, în grafurile ce vor fi distribuite între acele organizaţii);
    - *Un vocabular RDF se deosebeşte de un vocabular XML* prin aceea că nu permite validare. Aici nu lucrăm cu noțiunea de "date invalide" (=valori ce nu respectă un tip de date), ci cu cea de "afirmaţii contradictorii" (afirmații ce contrazic alte afirmaţii, de obicei din alt graf, altă proveniență).

3.2 Sintaxele Turtle și N-triples

Exemplu de afirmaţie (un graf e un set de astfel de afirmaţii):

```
@prefix : <http://buchmann.ro#>.
:Mary :fiicaLui :John.
```

[27] https://www.w3.org/RDF/
[28] https://www.w3.org/TR/sparql11-overview/
[29] https://www.w3.org/TR/rdf-schema/
[30] https://www.w3.org/OWL/

De reținut:
- Spre deosebire de XML, afirmaţiile RDF se pot scrie în mai multe sintaxe, fiecare având propriile reguli de delimitare ("bună formare"). Acest exemplu e scris în sintaxa Turtle[31] (vom studia pe parcurs şi alte sintaxe).
- Primul **termen** este *subiectul*, al doilea este *proprietatea (predicatul)*, al treilea este *obiectul* (dar nu în sensul programării obiectuale!)
- Fiecare "cuvânt" (termen) dintr-o afirmaţie are un prefix (în acest caz, caracterul ":"), pentru a indica cine "a inventat" termenul (pe prima linie, prefixul e asociat cu adresa de domeniu Web deţinută de creatorul termenilor).

O altă sintaxă pe care o vom folosi este N-triples[32]. Afirmaţia de mai sus poate fi scrisă în N-triples în felul următor:

```
<http://buchmann.ro#Mary> <http://buchmann.ro#fiicaLui> <http://buchmann.ro#John> .
```

În această sintaxă se vede mai clar că fiecare termen e precedat de "provenienţa" sa, indicată prin adresa de domeniu deținută de "creatorul" termenului. Faptul că orice adresă de domeniu Web are un proprietar unic, clar identificabil, garantează posibilitatea de a verifica provenienţa termenilor și de a solicita explicații privind semnificația lor. Principiul fundamental este:

- acelaşi termen să nu fie folosit cu mai multe semnificaţii (**http://buchmann.ro#Mary** nu are voie să reprezinte două persoane diferite în afirmaţii diferite, nici măcar din baze de grafuri diferite)
- în schimb pot exista mai mulţi termeni care să aibă aceeaşi semnificaţie (pe lângă **http://buchmann.ro#Mary** ar putea exista, cu referire la aceeaşi persoană, şi **http://anaf.ro#PopMaria** alocat de deţinătorii serverului anaf.ro); de obicei astfel de "termeni sinonimi" au provenienţă (adresă de domeniu) diferită

Cu alte cuvinte, *sinonimele sunt permise, însă omonimele NU* (pentru sinonime se pot construi dicţionare de echivalare care să permită oricui să verifice dacă doi termeni diferiţi se referă la acelaşi lucru).

Oricine deţine o adresă de domeniu poate inventa proprii termeni, dar poate şi adopta termeni inventaţi deja de altcineva. Putem combina în aceeaşi afirmaţie termeni de provenienţe diferite:

```
<http://anaf.ro#PopMaria> <http://schema.org/parent> <http://buchmann.ro#John> .
```

Aici am făcut aceeaşi afirmaţie ca mai sus, folosind termeni din **terminologii** diferite:

- Subiectul e un identificator universal (URI) alocat de proprietarii anaf.ro unei anume Pop Maria (practic devine un fel de CNP acordat de ANAF pentru scopuri proprii organizaţiei)
- Proprietatea **parent** aparţine terminologiei standard Schema.org (o colecţie standard de termeni recomandaţi de Google, Microsoft, Yahoo etc. pe care motoarele lor de căutare îi pot recunoaşte aducând diverse beneficii, inclusiv legate de SEO). Observaţi că delimitarea termenului de adresa de provenienţă se poate face şi cu slash, nu doar cu # (practic modurile de construire a unei adrese URL se pot folosi şi la a construi aceşti termeni)
- Obiectul John e un alt identificator universal (URI) al unei persoane, alocat de data asta de către deţinătorul serverului buchmann.ro.

Atunci când combinăm termeni din provenienţe diferite, în sintaxa Turtle trebuie să definim prefixe diferite pentru a distinge între provenienţe:

```
@prefix a: <http://anaf.ro#>.
@prefix : <http://buchmann.ro#>.
@prefix s: <http://schema.org/>.
a:PopMaria s:parent :John .
```

[31] https://www.w3.org/TR/turtle/
[32] https://www.w3.org/TR/n-triples/

Termenii nu trebuie scrişi într-o limbă anume – termenii creaţi de noi pot fi în orice limbă, inclusiv o limbă inventată sau un mod de codificare (dar atunci sunt mai greu de citit de către om). Termenii standard (precum s:parent din acest exemplu) au o formă în limba engleză. *În general nu contează limba, ci respectarea regulilor sintactice și reutilizarea corectă a termenilor (dacă același termen e folosit în mai multe afirmații, trebuie scris la fel de fiecare dată, cu aceeași adresă de proveniență).*

Toţi aceşti termeni se mai numesc **URI** (identificatori universali), însă chiar dacă arată ca nişte adrese URL, trebuie înţeleşi ca fiind "cuvinte" cu care formăm propoziţii:

- Dacă se referă la fiinţe, locuri etc., pot fi consideraţi ca nişte nume proprii sau *identificatori* precum CNP-ul (dar asigură o identitate mai largă, globală, în tot Web-ul);
- Dacă e vorba de adjective, verbe etc. vor fi considerate pur şi simplu "cuvinte" cu care formăm propoziții.

Nu toţi termenii din afirmaţii vor fi URI! Mai există două categorii de termeni: valorile literale şi nodurile anonime.

3.3 Afirmații cu valori literale

Valorile literale sunt orice fel de date, de orice tip (string, integer, boolean etc.) pe care dorim să le includem în afirmaţii, însă ele pot apare doar pe ultima poziţie! Valorile literale nu vor avea prefix/provenienţă:

Exemplu Turtle[33]:

```
@prefix : <http://buchmann.ro#>.
:Anna :hasAge 20 .
```

În sintaxa N-triples valorile literale apar ca simple stringuri adnotate cu tipul care le corespunde (tipul se preia din ierarhia standard de tipuri oferită de XML Schema):

```
<http://buchmann.ro#Anna> <http://buchmann.ro#hasAge> "20"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Practic valorile literale sunt ceea ce am stoca în baze de date relaţionale obişnuite – adică date de diverse tipuri. Un tabel cu vârstele și numele unor persoane ar putea fi tradus astfel în Turtle:

ID	Nume	Varsta
:PopAna	Pop Ana	20
:PopAndrei	Pop Andrei	30
:PopescuVasile	Popescu Vasile	50



```
@prefix : <http://buchmann.ro#> .
:PopAna :varsta 20; :nume "Pop Ana".
:PopAndrei :varsta 30; :nume "Pop Andrei".
:PopescuVasile :varsta 50; :nume "Popescu Vasile".
```

Observați folosirea delimitatorului ; pentru a grupa mai multe afirmații despre același subiect.

Remarcați distincția între **URI** (identificatori universali, jucând rolul de "chei primare" valabile în tot Web-ul) și **denumirile/numele** ale acelor lucruri/persoane (incluse ca stringuri). Nu pot exista două persoane/lucruri cu același URI (în tot Web-ul, nu doar în acest tabel!), dar pot exista două persoane/lucruri cu același nume.

Ați putea evita confuziile între URI și stringuri/denumiri, dacă în loc de URI folosiți niște coduri similare ID-urilor din baze de date relaționale...

```
@prefix : <http://buchmann.ro#>.
:P1 :varsta 20; :nume "Pop Ana".
```

[33] Observați spațiul dintre numărul 20 și punctul final. În absența acestui spațiu se va interpreta că vreți să scrieți un număr cu zecimale la care ați uitat să îi mai scrieți zecimalele iar unele sisteme vor semnala o eroare sintactică. Obișnuiți-vă să lăsați un spațiu înainte punctului final pentru a evita astfel de situații.

```
:P2 :varsta 30; :nume "Pop Andrei".
:P3 :varsta 50; :nume "Popescu Vasile".
```

...dar în această formă afirmațiile devin mai greu de citit de către om. Din acest motiv vom folosi termeni URI destul de apropiați de denumirea reală a lucrului pe care îl reprezintă, însă nu pierdeți niciodată din vedere distincția între URI și nume/denumiri:

- Denumirile sunt ceea ce trebuie să se vadă în interfața utilizatorului final (acesta nu trebuie să vadă URI, să nu simtă dacă în spatele unei pagini Web sunt tabele, grafuri, JSON sau altceva); din acest motiv, obișnuiți-vă să alocați un nume (de tip string) *oricărui subiect despre care se vor afișa informații în interfața cu utilizatorul*;
- Rolul termenilor URI nu e să fie afișați, ci să permită construirea de conexiuni complexe între lucruri/indivizi. Interogările vor naviga prin acele conexiuni, dar în final ele trebuie să afișeze spre utilizatorul final doar date (string, numeric, date calendaristice etc.).

E important să observați și modul în care se convertesc tabele relaționale în grafuri/afirmații:

- Valorile cheii primare devin subiecte în afirmații;
- Numele câmpurilor (capul de tabel, în afară de cheia primară) devin predicate (proprietăți);
- Toate datele (cu excepția cheii primare) devin obiecte ale afirmațiilor.

Diferența esențială este că aici, pe lângă datele din "tabel", fiecare subiect mai poate fi implicat în relații complexe (mai complexe decât ce permit cheile străine în bazele de relaționale) cu alte subiecte (din același "tabel" sau din altele, de pe același server sau de pe altele):

```
@prefix : <http://buchmann.ro#>.
@prefix alt: <http://altserver.ro#>.
:PopAna  :fiicaLui :PopIon .
:PopAndrei :traiesteLa alt:Cluj .
alt:Cluj alt:areEchipaDeFotbal alt:CFRCluj .
etc.etc. ...............................
```

3.4 Afirmații cu noduri anonime

Nodurile anonime sunt termeni cărora nu li s-a alocat încă un URI (fie că nu li se cunoaşte identitatea, fie că nu prezintă interes identitatea lor, fie că sunt noduri artificial introduse ca să conectăm mai multe afirmaţii într-un graf comun). Pentru nodurile anonime, în loc de URI se folosesc variabile prefixate cu underscore. Afirmaţiile de mai jos se traduc prin "Ana cunoaşte pe cineva care cunoaşte pe Petru":

```
@prefix : <http://buchmann.ro#>.
:Anna :cunoastePe _:x .
_:x :cunoastePe :Petru .
```

SAU, putem folosi o exprimare mai concisă, fără a mai include acea "variabilă" _:x :

```
@prefix : <http://buchmann.ro#>.
:Anna :cunoastePe [:cunoastePe :Petru] .
```

3.5 Conversii sintactice

Accesați următorul instrument online pentru a realiza conversii RDF între diferite sintaxe:
http://www.easyrdf.org/converter

Cunoștințele de introdus:
Anna are culoarea părului roșie, are vârsta de 20 și este fiica lui John. Mary este, de asemenea, fiica lui John.
Puteți folosi orice editor text (Notepad) pentru sintaxa Turtle[34]:

[34] La fel de bine am putea scrie şi în forma :Anna :areVarsta 20; :eFiicaLui :John. Mai mult, afirmaţiile se scriu cu setul de caractere UNICODE, deci pot conţine şi diacritice, litere chinezeşti etc. Se preferă totuşi termeni care să poată fi citiţi uşor de orice utilizator Web din lume.

```
@prefix : <http://expl.at#>.
:Anna :hasHairColor :Red; :hasAge 20; :daughterOf :John.
:Mary :daughterOf :John.
```

- Să se copieze textul în secțiunea Input Data a convertorului EasyRDF
- Se va selecta formatul de introducere (Input Format): Turtle
- Încercați cu diferite opțiuni pentru formatul de ieșire (Output Format) pentru a observa cum arată alte sintaxe RDF
- Puteți să folosiți acest instrument și ca validator, pentru a verifica dacă ați introdus declarațiile corect în sintaxa Turtle: dacă reușiți să îl convertiți fără erori înseamnă că este corect

Sintaxele RDF sunt:
• Turtle, TriG[35] (simplu pentru citit/scris, cel mai important)
• N-triples, N-quads[36] (simplu pentru citit, dificil de tastat)
• RDF / XML[37] (dificil de citit și scris, dar standardizat și suportat de orice software)
• TriX[38] (ușor de citit, dificil de scris, bazat pe XML)
• RDFa[39] (dificil de citit și scris, dar integrabil în codul HTML al paginilor Web)
• JSON-LD[40] (dificil de citit și scris, dar integrabil în codul JavaScript al paginilor Web)

EasyRDF converter oferă, de asemenea, unele formate de ieșire grafice (PNG, GIF), care ar trebui să genereze o reprezentare grafică sub forma unui graf. Dar uneori aceste opțiuni vor da o eroare în varianta on-line (puteţi descărca pachetul ca librărie PHP!).

Puteți încerca, de asemenea, instrumentul on-line pentru validarea și vizualizarea sintaxei RDF / XML:
http://www.w3.org/RDF/Validator/
(dar trebuie să generați sintaxa RDF/XML din sintaxa Turtle (cu easyRDF), pentru că RDF/XML este singurul tip de intrare acceptat de acest validator).

O reprezentare grafică arată foarte clar de ce cunoștințele sunt grafuri și de ce limbajul XML sau tabelele din baze de date relaţionale nu sunt suficiente pentru baze de cunoștințe. Exemplul nostru are următoarea formă a grafului*:*

[35] https://www.w3.org/TR/trig/
[36] https://www.w3.org/TR/n-quads/
[37] https://www.w3.org/TR/rdf-syntax-grammar/
[38] http://www.hpl.hp.com/techreports/2004/HPL-2004-56.html
[39] https://www.w3.org/TR/rdfa-primer/
[40] https://www.w3.org/TR/json-ld/

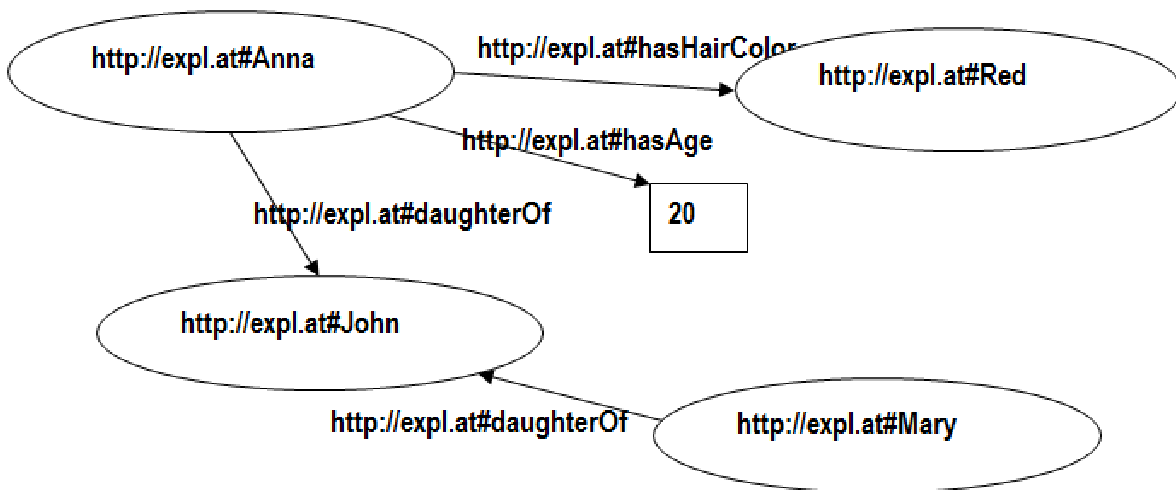


Figura 28 Reprezentarea grafică pentru grafurile de cunoștințe (exemplul 1)

Traduceți următorul graf în limbaj natural și în Turtle

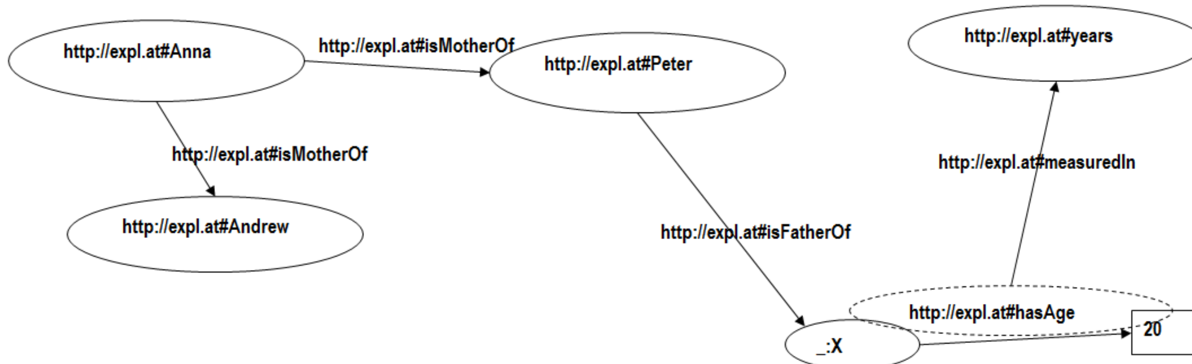


Figura 29 Reprezentarea grafică pentru grafurile de cunoștințe (exemplul 2)

Turtle:

```
@prefix : <http://expl.at#> .
:Anna :isMotherOf :Andrew, :Peter.
:Peter :isFatherOf [:hasAge 20].
:hasAge :measuredIn :years.
```

Limbaj natural:

Anna este mama lui Andrew și Peter. Peter este tatăl cuiva a cărui vârstă este 20. Vârsta se măsoară în ani.

Comentarii: Observaţi că printre termenii folosiţi mai apar: o valoare simplă (numărul 20) și un nod anonim (nod blank, ceva despre care noi nu știm nimic altceva, sau nu ne interesează suficient încât să-i alocăm un identificator) . În N-triples acest lucru se traduce prin:

```
<http://expl.at#Anna> <http://expl.at#isMotherOf> <http://expl.at#Andrew> .
<http://expl.at#Anna> <http://expl.at#isMotherOf> <http://expl.at#Peter> .
_:x <http://expl.at#hasAge> "20"^^<http://www.w3.org/2001/XMLSchema#integer> .
<http://expl.at#Peter> <http://expl.at#isFatherOf> _:x .
<http://expl.at#hasAge> <http://expl.at#measuredIn> <http://expl.at#years> .
```

Comentarii:

• Pentru nodurile anonime nu putem folosi identificatori universali (URI). În schimb folosim o construcţie precum _: x (s-ar putea să aveți altceva în loc de "x", pentru că e vorba de un identificator local generat aleator în timpul conversiei de sintaxă). Ceea ce înseamnă că nu îl putem refolosi (se poate modifica/regenera de fiecare dată când salvați aceste afirmaţii într-o bază de cunoștințe!).

• Pentru valori simple (de exemplu, numere), nu vom folosi un URI, trebuie doar să îi atribuim un tip de date din XML Schema, care la rândul lor au formă de URI).

3.6 Vocabulare RDF

Pornim de la următorul graf:

```
@prefix : <http:/buchmann.ro#>.
:Anna :hasHairColor :Red; :hasAge 20; :daughterOf :John.
:Mary :daughterOf :John.
```

Aceasta e o bază de date de tip schemaless ("fără schemă","fără vocabular"), noțiune care nu există în bazele de date relaționale (acolo nu putem începe introducerea de date fără a defini în prealabil schema/structura bazei de date). O bază de date schemaless e totuși utilă, pentru că poate fi interogată, însă cu anumite limitări:

- poate fi interogată doar de creatorul său care știe ce înseamnă și cum se scriu termenii folosiți[41]; aceasta poate fi suficient în cazul unei baze de date închise, izolate (când cel care a creat baza de date și cel care o va interoga e aceeași persoană/echipă);
- se vor putea realiza interogări de genul "ce vârstă are Anna?" dar nu și interogări de genul "afișează lista tuturor persoanelor"; pentru al doilea tip de interogări e necesar un **vocabular RDF** (numit și terminologie, ontologie, taxonomie, schemă RDF) care să stabilească CE ESTE fiecare individ/lucru menționat în afirmații.

Sarcina de bază pe care trebuie să o îndeplinească un vocabular RDF este să permită răspunsul la întrebarea "Ce este X?" (ce tip are X?) pentru orice URI dintr-un graf (dacă se garantează acest lucru, înseamnă că se poate formula și interogarea "Afișează lista cu toți X").

Pentru ca acest lucru să fie posibil trebuie realizate două cerințe:

1. un vocabular va oferi lista completă de clase (tipuri) și de proprietăți (predicate) folosite în graf, sub forma unor **declarații de clase și proprietăți**, realizate cu ajutorul unor termeni standardizați (prefixați cu rdf: și rdfs, propuși de consorțiul W3C):

```
@prefix v: <http:/buchmann.ro/vocabularulMeu#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
v:daughterOf a rdf:Property; rdfs:comment "reprezintă relația de fiică (a unei alte persoane)".
v:hasHairColor a rdf:Property; rdfs:comment "reprezintă atributul culoare de păr".
v:hasAge a rdf:Property; rdfs:comment "reprezintă atributul vârstă, exprimat în ani".
v:Color a rdfs:Class; rdfs:comment "reprezintă mulțimea tuturor culorilor".
v:Man a rdfs:Class; rdfs:comment "reprezintă mulțimea tuturor bărbaților".
v:Woman a rdfs:Class; rdfs:comment "reprezinta mulțimea tuturor femeilor".
```

2. apoi, în graf vor trebui incluse afirmații de forma "X este un Y" pentru orice termen X

```
@prefix: <http:/buchmann.ro#>.
@prefix v: <http:/buchmann.ro/vocabularulMeu#>.
:Anna a v:Woman.
:Red a v:Color.
:John a v:Man.
:Mary a v:Man.
:Anna v:hasHairColor :Red; v:hasAge 20; v:daughterOf :John.
:Mary v:daughterOf :John.
```

Observați termenii standardizați:

- litera "a" neprefixată reprezintă verbul "a fi" (se poate folosi și forma prefixată *rdf:type*)

[41] Noțiunea de bază de date schemaless există și în XML sau JSON, pentru că și acolo putem crea sau interoga date chiar și în absența unui vocabular.

- rdf:Property reprezintă noțiunea de proprietate
- rdfs:Class reprezintă noțiunea de tip (clasă, mulțime de indivizi)
- rdfs:comment e o proprietate standard ce permite să atașăm oricărui termen o descriere textuală a semnificației sale; aici am inclus descrieri în limba română, dar se pot atașa descrieri în mai multe limbi, sau doar în engleză - în funcție de publicul țintă care trebuie să înțeleagă semnificația termenilor (se recomandă totuși engleza).

Comentarii cu privire la conținutul vocabularului:

- observați că am diferențiat prin prefixe *termenii definiți în vocabular* (v:Man, v:hasAge etc.) de *identificatorii ce reprezintă indivizi/instanțe individuale* (:Anna, :John); acest lucru nu e obligatoriu dar e recomandat în practică, în speranța că partea de vocabular va fi reutilizată și de alții (totuși în exercițiile noastre vom avea de regulă același prefix);
- partea de vocabular se poate stoca separat pentru a fi consultată de oricine (precum vocabularele XML) sau se poate adăuga în aceeași bază de date;
- vocabularul e un fel de **dicționar de termeni** (de aceea e numit și "terminologie"), însă poate conține mult mai mult decât atât, după cum vom studia în viitor;
- vocabularul respectă sintaxa RDF, deci e tot o colecție de afirmații (așadar e tot un graf, poate fi interogat, combinat cu alte grafuri etc.); diferența între vocabular și restul afirmațiilor nu e dată de sintaxă, ci de termenii folosiți:
    - o în general vocabularele conțin afirmații ale căror subiecte sunt fie tipuri (clase), fie proprietăți, nu și instanțe/indivizi;
    - o afirmațiile din vocabular folosesc o serie de termeni standardizați, ce permit clienților și instrumentelor software să le proceseze în anumite moduri standard.

Adăugând toate acestea la afirmațiile inițiale, *am transformat un graf schemaless într-o bază de cunoștințe completă*.

Exemplu:
Traduceți în limbaj natural, în sintaxă Turtle cu "termenii" vizibili în imagine, apoi în sintaxă Turtle cu termeni standardizați:

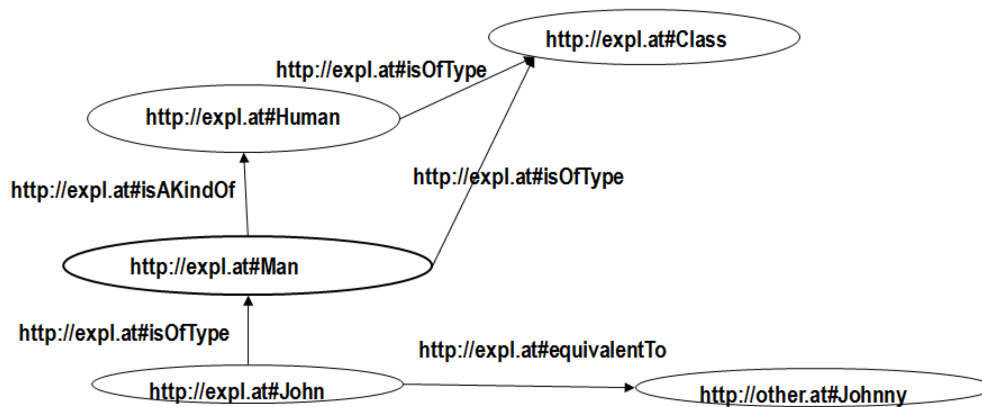


Figura 30 Reprezentarea grafică pentru grafurile de cunoștințe (exemplul 3)

Limbaj natural: John este bărbat. Orice bărbat este un om. Bărbații și oamenii sunt mulțimi. John este, de asemenea, cunoscut şi sub identificatorul Johnny (propus de cei care deţin serverul http://other.at)

Sintaxă Turtle cu termenii din imagine:

```
@prefix : <http://expl.at#> .
@prefix o: <http://other.at#> .
:John :isOfType :Man; :equivalentTo o:Johnny.
:Man :isAKindOf :Human; :isOfType :Class.
:Human :isOfType :Class.
```

Ultimele două linii formează vocabularul pentru prima linie. Nu e eronat din punct de de vedere sintactic să construim vocabulare și cu proprii noștri termeni însă, pentru a ne asigura că oricine va putea interoga și interpreta vocabularul, se recomandă să învățăm termenii standardizați pentru diverse concepte cheie (verbul "a fi", noțiunea de proprietate etc.) și să construim vocabularele cu aceștia:

```
@prefix : <http://expl.at#> .
@prefix o: <http://other.at#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns# > .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema# > .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
:John a :Man; owl:sameAs o:Johnny.
:Man rdfs:subClassOf :Human; a rdfs:Class.
:Human a rdfs:Class.
```

Observați termenii standard nou introduși:

- **owl:sameAs** reprezintă relația de echivalență (sinonimie) prin care putem exprima faptul că un același individ are mai mulți URI (de obicei de proveniență diferită); reamintim că e posibil ca un lucru să aibă mai multe identități, dar nu e permis ca o identitate să fie folosită pentru mai multe lucruri;
- **rdfs:subClassOf** poate fi înțeles din mai multe puncte de vedere:
    - o dpdv lingvistic este verbul "a fi" la plural ("bărbați sunt oameni"),
    - o dpdv matematic e relația de incluziune între mulțimi (mulțimea bărbaților e inclusă în mulțimea oamenilor)
    - o dpdv logic e o implicație ("orice bărbat este și om", sau "dacă X e bărbat rezultă că X e om")
    - o dpdv obiectual e o "moștenire" (clasa bărbaților o specializează/moștenește pe cea a oamenilor)

Exemplu: Traduceți următorul text în Turtle (puteți desena graful mai întâi)

Capitala Austriei este Vienna sau Innsbruck. Austria este o țară, Vienna și Innsbruck sunt orașe. Există o femeie care trăiește pe strada Brunnerstrasse, care este o stradă din Vienna.

```
@prefix : <http://expl.at#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns# > .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema# > .
:Austria :hasCapital [a rdf:Alt; rdfs:member :Vienna, :Innsbruck]; a :Country.
:Vienna a :City.
:Innsbruck a :City.
[a :Woman; :livesOn :Brunnerstrasse].
:Brunnerstrasse a :Street; :locatedIn :Vienna.
```

Observați două moduri de utilizare a nodurilor invizibile:

- pentru un individ neidentificat/neidentificabile ("există" o femeie neidentificată)
- ca termen artificial introdus pentru a construi o frază complexă (cu ajutorul unor termeni standardizați):
    - o rdf:Alt indică faptul că urmează să se enumere mai mulți termeni care trebuie percepuți ca alternative (cu SAU între ei)
    - o rdfs:member e relația prin care conectăm lista termenilor între care se aplică acel SAU

Exemplu: Traduceți în Turtle

Vara are următoarele luni, în această ordine: iunie, iulie, august.
James Cameron a regizat Avatar, Aliens și Terminator (ordinea e irelevantă).

```
@prefix : <http://expl.at#> .
:Summer :hasMonths (:June :July :August).
:JamesCameron :directed :Avatar, :Aliens, :Terminator.
```

Observați două moduri de enumerare a mai multor obiecte ce au aceeași relație cu același subiect:

- *enumerarea inclusă în paranteze rotunde, cu spațiu între obiecte* înseamnă că ordinea contează și acea ordine nu e una sintactică/alfabetică, ci ține de semnificația termenilor
    - => se va putea interoga prima lună vară, ultima lună de vară, luna de după Iunie etc.
- *enumerarea fără paranteze, cu virgulă între termeni* înseamnă că ordinea nu contează
    - => se va putea obține lista completă a filmelor lui James Cameron dar nu și primul film, ultimul film etc. (eventual se poate face maxim o ordonare după criterii sintactice – de exemplu o listare alfabetică)

3.7 Gruparea afirmațiilor în mai multe grafuri

Uneori doriți să grupați afirmaţii. Le puteți grupa în grafuri separate (aşa cum în bazele de date relaţionale grupăm datele în tabele diferite). Următorul exemplu conține 2 grafuri (când folosim grafuri, sintaxa este numită **TriG** – o extensie a lui Turtle, pentru că iniţial standardul Turtle nu permitea această grupare):

```
@prefix x: <http://expl.at#>.
@prefix o: <http://other.at#>.

x:Graph1
{
x:John a x:Human; x:isBrotherOf x:George, x:Anna; x:fullName "John Smith".
x:Anna a x:Human.
x:George a x:Human.
}
o:Graph2
{o:Mary o:worksAt [a o:Company; o:locatedIn o:Wien]}
```

Tradus în limbaj natural:
Graf-ul 1 (creat de deţinătorii serverului expl.at): John este om, este fratele lui George și al Anei, iar numele său complet este "John Smith". Anna este om, George este om.
Graf-ul 2 (creat de deţinătorii serverului other.at): Mary lucrează la o companie care este situată în Viena.

Următorul exemplu este un singur graf:

```
@prefix x: <http://expl.at#>.
x:Graph1
{
x:Mary x:worksAt x: ABC; x:motherOf x:Peter, x:Andrew; x:hasAge 40 .
x: ABC a x:Company; x:locatedIn x:Wien .
x:Andrew x:isBrotherOf x:Peter; x:hasAge 20 .
}
```

Tradus în limbaj natural:
Graf 1: Mary lucrează la ABC, este mama lui Peter și Andrew, are vârsta de 40. ABC este o companie și este situată în Viena. Andrew este fratele lui Peter, și are 20 de ani.

4. Interogarea grafurilor de cunoștințe prin SPARQL

4.1 Interogări SPARQL in RDF4J

În cele ce urmează vom folosi serverul de baze de grafuri RDF4J pentru a exersa interogări pe grafuri RDF. Instrumente necesare:

- Conexiune internet
- Tomcat 8.0 (necesită ca Java să fie instalat corect)
- RDF4J (necesită ca Java să fie instalat corect)

Descărcați RDF4J de la adresa de mai jos:
http://rdf4j.org/download/

Descărcați Tomcat la adresa *http://tomcat.apache.org/download-80.cgi* (folosiţi Tomcat 8.0). Descărcaţi varianta ZIP (nu service installer).

Copiați conținutul directorului War din pachetul Eclipse-rdf4j (2 fișiere) în directorul webapps din Tomcat. Porniți serviciul Tomcat din *Tomcat\bin\startup.bat*.

Asigurați-vă că Java e instalat și configurat corect. Dacă fereastra startup.bat se închide imediat după ce porneşte înseamnă de obicei că:

- Java nu e instalat, SAU...
- că e instalat dar nu s-a setat variabila de mediu JRE_HOME (dacă ați instalat varianta Java JRE) sau variabila JAVA_HOME (dacă aţi instalat varianta Java JDK). Una din cele 2 variabile trebuie să existe în variabilele de mediu Windows (Advanced System Settings-Environment Variables-System Variables). Valoarea variabilei trebuie să fie locația la care ați instalat Java - de exemplu C:\Program Files\Java\jdk1.8.0_77 (acesta e cazul variantei JDK instalată pe un Windows 7 64 bit – locația va fi diferită pe alte versiuni de Windows sau dacă aveți varianta Java JRE)

După ce v-ați asigurat că fereastra de pornire Tomcat rulează, o puteți minimiza (închiderea sa va opri serverul Tomcat – obișnuiți-vă totuși să închideți serverul cu fișierul shutdown.bat, deoarece oprirea bruscă a ferestrei produce uneori pierderi de date).

Creați o bază de cunoștințe RDF4J prin interfața Web Workbench

Porniți browser-ul și lansați aplicația Web RDF4J la adresa: *http://localhost:8080/rdf4j-workbench*. În panoul din stânga al interfeței RDF4J selectați *Repositories-New repository* cu următorii parametri:

- *Type = Native Java Store*
- *ID = MyRepo*
- *Title = MyRepo*
- *Triple indexes = spoc, posc*

Salvați următoarele date în fișierul *movies.trig.*

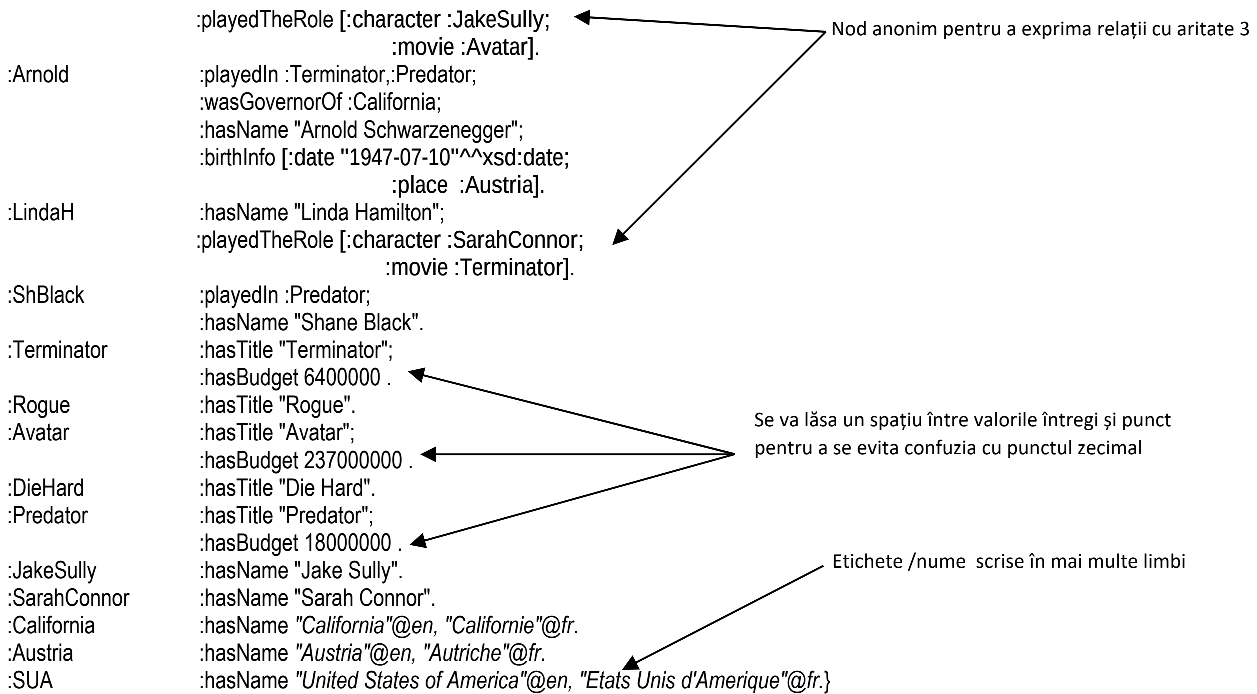
```
@prefix : <http://expl.at#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
:mymoviegraph
{:JamesCameron    :directorOf :Avatar,:Terminator;
                  :hasName "James Cameron".
:JohnMcT          :directorOf :Predator, :DieHard;
                  :hasName "John McTiernan";
                  :birthInfo _:birthdetails.
_:birthdetails    :date "1951-01-08"^^xsd:date;
                  :place  :SUA.
:McG              :directorOf  _:somemovie;
                  :hasName "Joseph McGinty";
                  :hasNickname "McG".
:SamWorth         :hasName "Sam Worthington";
                 :playedIn :Rogue, _:somemovie;
```

Toate datele sunt grupate în acest graf identificat

Afirmații multiple despre același subiect și aceeași proprietate

Noduri anonime ca structuri de date

Noduri anonime folosite pentru a ține locul unei resurse neidentificate



```
                  :playedTheRole [:character :JakeSully;
                                  :movie :Avatar].
:Arnold           :playedIn :Terminator,:Predator;
                  :wasGovernorOf :California;
                  :hasName "Arnold Schwarzenegger";
                  :birthInfo [:date "1947-07-10"^^xsd:date;
                              :place  :Austria].
:LindaH           :hasName "Linda Hamilton";
                  :playedTheRole [:character :SarahConnor;
                                  :movie :Terminator].
:ShBlack          :playedIn :Predator;
                  :hasName "Shane Black".
:Terminator       :hasTitle "Terminator";
                  :hasBudget 6400000 .
:Rogue            :hasTitle "Rogue".
:Avatar           :hasTitle "Avatar";
                  :hasBudget 237000000 .
:DieHard          :hasTitle "Die Hard".
:Predator         :hasTitle "Predator";
                  :hasBudget 18000000 .
:JakeSully        :hasName "Jake Sully".
:SarahConnor      :hasName "Sarah Connor".
:California       :hasName "California"@en, "Californie"@fr.
:Austria          :hasName "Austria"@en, "Autriche"@fr.
:SUA              :hasName "United States of America"@en, "Etats Unis d'Amerique"@fr.}
```


Observaţi că fişierul e scris în sintaxa TriG (are cunoştinţele grupate între acolade, într-un "graf identificat"). În RDF4J, grafurile identificate se mai numesc şi **contexte**.

Încărcați fișierul în RDF4J, folosindu-vă de Modify/Add[42] și următoarele opțiuni:

- *RDF Data File* = căutați fișierul movies.trig
- *Data format* = TriG
- Câmpul *Context* ar trebui să rămână liber deoarece contextul (:mymoviegraph) este luat direct din fișier; dacă se încarcă un fișier fără graf identificat, atunci contextul poate fi introdus manual);
- Debifați checkboxul *Use base URI as context*, din același motiv
- Apăsați *Upload*.

Puteți converti datele în sintaxa dorită cu opțiunea *Export* (în cazul în care selectați un format care nu suportă grafuri identificate precum N-Triples, Turtle, identificatorul contextului nu va fi exportat). În fereastra *Export*, puteți da clic pe oricare din URI-uri pentru a extrage toate afirmațiile despre acel URI. Observați cum arată nodurile anonime – nu mai sunt la fel ca în fișierul original! (reamintim: nodurile anonime au identificatori instabili ce nu pot să apară în interogări).

Executați o primă interogare în ecranul *Query*:

```
select * where {?x ?y ?z}
```

Aceasta va afişa toate afirmaţiile din baza de cunoștințe:

Executați câteva interogări SPARQL generice (fără semantică)

Extragerea tuturor subiectelor din baza de cunoștințe:

```
select ?x where {?x ?y ?z}
```

Extragerea de subiecte distincte din baza de cunoștințe:

```
select distinct ?x where {?x ?y ?z}
```

Extragerea de afirmații ordonate după proprietate:

[42] Uneori e posibil ca meniul din stânga paginii să fie inactiv imediat după crearea unei baze de cunoștințe noi (de obicei în Chrome). În acest caz faceți un Refresh cu tasta Shift apăsată (e o problemă de cache a browserului)

```
select * where {?x ?y ?z} order by ?y
```

Extragerea primelor 10 afirmații ordonate după subiect:

```
select * where {?x ?y ?z} order by ?x limit 10
```

Executați câteva interogări SPARQL cu semantică simplă

La folosirea identificatorilor de tip URI într-o interogare, toate prefixele trebuie să fie declarate înainte de interogare . Pentru a face acest lucru mai ușor, procedați în felul următor :

- Verificați opțiunea *Namespaces* din meniul situat în partea stângă a interfeței RDF4J. Toate prefixele care au fost detectate în fișierul original ar trebui să apară listate acolo. De asemenea, se pot adăuga altele noi;
- Dacă începeți să tastați "prefix" în fereastra *Query* și veți obține o listă derulantă de prefixe existente (lista de la *Namespaces*) pentru a alege pe cele de care aveți nevoie în interogarea curentă.
- Dacă tastați un prefix unele versiuni inserează automat declarația acelui prefix (dacă e dintre cele prezente în fișierul original).

În alte sisteme e posibil să nu aveţi această facilitate (inclusiv versiuni mai vechi de RDF4J) şi atunci trebuie să includeţi prefixele manual. Pentru fiecare interogare, trebuie să vă asigurați că prefixele sunt incluse în fața interogării după cum arată primul exemplu de mai jos (în restul exemplelor, prefixul va fi omis în acest document, dar voi trebuie să îl aveţi inclus de fiecare dată):

În ce filme a jucat Arnold?

```
prefix : <http://expl.at#>
select ?x
where
{
:Arnold :playedIn ?x
}
```

Cu ce are Arnold alte relaţii decât :playedIn?

```
select ?x
where
{
:Arnold !:playedIn ?x
}
```

(avertisment: a nu se confunda cu negația - nu se vor obține filmele în care NU a jucat Arnold! Veți obține toate obiectele cu care Arnold are altă relație decât :playedIn)

Afișează tot ce este în relație cu Terminator:

```
select ?x ?y
where
{
?x ?y :Terminator
}
```

Afișează tot ce este în relație cu Terminator și în plus afișează Terminator pentru fiecare rezultat:

```
select ?x ?y (:Terminator as ?z)
where
{
?x ?y :Terminator
}
```

Afișează tot ceea ce are nume, împreună cu numele acestora:

```
select ?x ?y
where
{
?x :hasName ?y
}
```

Executați câteva interogări SPARQL cu filtre

Aceeași interogare precum cea anterioară, dar să se afișeze doar acele rezultate care au numele disponibil în franceză:

```
select ?x ?y
where
{
?x :hasName ?y
filter (lang(?y)="fr")
}
```

Afișează tot ce are buget mai mic decât 10000000:

```
select ?x ?b
where
{
?x :hasBudget ?b
filter (?b<10000000)
}
```

Cine are ziua de naștere 1947-07-10?

```
select ?x ?d
where
{
?x :birthInfo/:date ?d
filter (?d="1947-07-10"^^xsd:date)
}
```

(trebuie să traversăm atât :birthInfo cât și :date datorită nodului anonim care grupează data și locul)

Cine este cel al cărui nume se termină în "on" (indiferent că e majusculă sau minusculă)?

```
select ?x
where
{
?x :hasName ?nm
filter regex(?nm,"ON$","i")
}
```

(regex permite să se aplice filtre cu expresii regulate!)

Care URI se termină în "on" (indiferent de tipul de literă)?

```
select ?x
where
{
?x :hasName ?nm
filter regex(str(?x),"ON$","i")
}
```

De această dată regex nu se aplică șirurilor de caractere (nume, etichete) ci identificatorilor (URI-urilor) iar rezultatele vor fi diferite. A se observa că identificatorii trebuie să fie convertiți în șiruri de caractere cu str pentru a li se aplica testul regex! Această tehnică este folositoare pentru a detecta anumiți identificatori care aparțin anumitor adrese de domeniu.

Să se afișeze toate afirmațiile pentru care obiectul este valoare simplă:

```
select *
where
{
?x ?y ?z
filter isliteral(?z)
}
```

Sunt disponibile funcții similare pentru verificarea dacă un termen este URI (*isuri*) sau dacă este nod anonim (*isblank*).

Să se afișeze toate afirmațiile care au Terminator ca subiect SAU ca obiect:
```
select *
where
{
?x ?y ?z
filter ((?x=:Terminator)||(?z=:Terminator))
}
```

Alți conectori logici disponibili sunt && (conjuncția) și ! (negația)

Executați câteva interogări SPARQL care returnează expresii

Să se construiască o afirmație în limbaj natural ce indică bugetul fiecărui film.
```
select (concat(?t," has a budget of ",str(?b)) as ?sentence)
where
{
?x :hasTitle ?t; :hasBudget ?b
}
```
(această tehnică poate fi folosită pentru a construi stringuri gata de afișat în cadrul unei interfețe cu utilizatorul!)

Să se extragă prenumele oricui are un nume:
```
select (strbefore(?y," ") as ?firstname)
where
{
?x :hasName ?y
}
```

Să se afișeze filmele cu bugetele lor, plus un comentariu indicând dacă este un buget mic (sub 10000000) sau mare:
```
select ?x (if(?b<10000000,"Low budget","Big budget") as ?comment)
where
{
?x :hasBudget ?b
}
```

Să se afișeze bugetul mediu:
```
select (avg(?b) as ?AverageBudget)
where
{
?x :hasBudget ?b
}
```
(alte funcții de agregare disponibile sunt: count, sum, min, max, group_concat(concatenare de șiruri), sample (selectare arbitrară dintr-o listă de valori))

Să se construiască un string în care se înşiră toate titlurile separate de virgulă:
```
select (group_concat(?t;separator=', ') as ?TitleList)
where
{
?x :hasTitle ?t
}
```

Să se afișeze toți regizorii și numărul de filme pe care le-au regizat:
```
select ?x (count(?m) as ?MovieCount)
where
{
?x :directorOf ?m
}
group by ?x
```
(avertisment: o interogare cu grupare poate afișa doar variabilele folosite în clauza GROUP BY (în cazul nostru ?x) sau variabile agregate (aici ?MovieCount))
Aceeași interogare precum cea anterioară, însă păstrând doar regizorii care au mai mult de un film:

```
select ?x (count(?m) as ?MovieCount)
where
{
?x :directorOf ?m
}
group by ?x
having (?MovieCount>1)
```

Să se afișeze câte un exemplu de film pentru fiecare regizor:

```
select ?x (sample(?m) as ?Example)
where
{
?x :directorOf ?m
}
group by ?x
```

Construiți o listă de titluri pentru fiecare regizor care are mai mult de un film:

```
select ?x (group_concat(?t;separator=', ') as ?TitleList)
where
{
?x :directorOf/:hasTitle ?t
}
group by ?x
having (count(?t)>1)
```

(Observați cum proprietățile :directorOf/:hasTitle sunt înlănțuite pentru a lega regizorii de titluri)

Executați interogări SPARQL cu subinterogări

Să se afișeze filmul cu bugetul minim.

```
select ?minbfilm ?minb
where
{
?minbfilm :hasBudget ?minb
{select (min(?y) as ?minb) where {?x :hasBudget ?y}}
}
```

Acest exemplu se execută în 2 pași:

- Prima dată (în subinterogare) se găsește bugetul minim și se păstrează într-o variabilă (?minb)
- Apoi (în interogarea principală) se caută filmul cu bugetul respectiv

Pentru a transfera rezultatul de la subinterogare la interogarea principală s-a folosit o variabilă comună (?minb).

Executați interogări SPARQL contextuale (cu grafuri identificate)

Pentru a putea realiza acest lucru, avem nevoie de cel puțin 2 contexte (grafuri identificate). Creați un nou fișier *movies2.trig* cu următorul conținut:

```
@prefix : <http://expl.at#>.
:mynewmoviegraph
{:Titanic           :directedBy         :JamesCameron;
                    :hasActor           :LeoDiCaprio.
:Godzilla           :directedBy         :REmmerich.
:DieHard            :hasActor           :BruceWillis.
:JamesCameron       :directorOf         :Terminator.
:Predator           :hasBudget          100 .
:Titanic            :hasBudget          200 .
:DieHard            :hasBudget          300 .
:Godzilla           :hasBudget          400 .
_:somemovie         :hasBudget          500 .
}
```

Observați următoarele:

- Relația regizor-film este acum exprimată în două moduri diferite (:directedBy și :directorOf). Deocamdată RDF4J nu înțelege că aceste relații sunt reciproc inverse! (e necesară utilizarea terminologiei standard OWL pentru a face posibil acest lucru);
- Am adăugat câteva afirmații care contrazic afirmațiile din graful inițial, de exemplu bugetul filmului Predator. Se va putea detecta această contradicție tot prin raționare;
- Deși s-a folosit un identificator pentru nod anonim _:somemovie când s-a încărcat primul fișier, ulterior a primit un nou ID și astfel s-a pierdut legătura cu afirmațiile anterioare (puteți vedea toate afirmațiile în ecranul *Export*).

Încărcați noul fișier în aceeași bază de cunoștințe și testați următoarele interogări:

Să se afișeze toate afirmațiile despre James Cameron, împreună cu graful de care aparțin:

```
select *
where
{
graph ?g {?x ?y ?z filter (?x=:JamesCameron)}
}
```

(observați că apare de două ori afirmația că James Cameron a regizat Terminator – acest lucru este permis deoarece e stocată în grafuri diferite; în cadrul aceluiași graf, o afirmaţie poate apare o singură dată!).

Să se afișeze toate afirmațiile din graful nou încărcat:

```
select *
from :mynewmoviegraph
where {?x ?y ?z}
```

Aceleași rezultate pot fi obținute prin formularea de mai jos:

```
select *
where
{
graph :mynewmoviegraph {?x ?y ?z}
}
```

Deosebiri:

- Cu clauza FROM, se va separa primul graf de restul cunoştinţelor și apoi se execută interogarea pe graf
- Cu clauza GRAPH se execută interogarea pe tot setul de date și apoi se filtrează rezultatele după graful indicat

În cele mai multe cazuri rezultatele sunt identice. Dacă se folosesc deodată, FROM are prioritate. Totuși se preferă clauza GRAPH deoarece permite interogări ce nu sunt posibile cu clauza FROM, precum:
- folosirea identificatorului de graf ca variabilă (să se extragă toate afirmațiile despre Predator, inclusiv numele grafului de care aparțin)

```
select *
where
{
graph ?g {?x ?y ?z}
filter (?x=:Predator)
}
```

-combinarea șabloanelor din grafuri diferite (să se extragă toate afirmațiile din graful iniţial, despre cel care a regizat Titanic conform grafului nou)

```
select *
where
{
graph :mynewmoviegraph {:Titanic :directedBy ?x}
graph :mymoviegraph {?x ?y ?z}
}
```

A se observa variabila comună ?x care realizează înlănțuirea afirmațiilor din diferite grafuri.

Să se compare ultimul rezultat cu următorul (toate afirmaţiile despre cel care a regizat Titanic, conform grafului nou):

```
select *
where
{
graph :mynewmoviegraph {:Titanic :directedBy ?x}
?x ?y ?z
}
```

… și cu următorul (toate afirmaţiile din noul graf, despre cel care a regizat Titanic conform aceluiaşi graf):

```
select *
where
{
graph :mynewmoviegraph {:Titanic :directedBy ?x. ?x ?y ?z}
}
```

Să se afişeze toate subiectele şi proprietăţile folosite împreună în ambele grafuri:

```
select distinct ?x ?y
where
{
graph :mynewmoviegraph {?x ?y ?z}
graph :mymoviegraph {?x ?y ?c}
}
```

Această tehnică poate fi folosită pentru a descoperi dacă aceeași proprietate are valori diferite (pentru același lucru) în contexte diferite. Se poate folosi:

- Pentru a agrega informații incomplete din surse multiple (presupunând că fiecare graf conține informații parțiale)
- Pentru a detecta contradicții (de exemplu verificând dacă aceeași proprietate, de exemplu ziua de naştere sau CNP-ul este declarat diferit în baze de cunoştinţe diferite, ceea ce poate indica un caz de fraudă!).

În cazul de faţă vom obţine două rezultate:

- faptul că James Cameron are relaţia directorOf inclusă în ambele grafuri; nu e nicio problemă cu asta, deoarece pe baza sa putem colecta o listă de filme din ambele surse;
- faptul că Predator are declarate bugete în ambele grafuri; aceasta poate fi o problemă, şi trebuie să verificăm dacă acele bugete sunt identice sau nu, cu o interogare de genul:

```
select distinct ?x ?z ?c
where
{
graph :mynewmoviegraph {?x :hasBudget ?z}
graph :mymoviegraph {?x :hasBudget ?c}
}
```

(interogarea nu caută doar pe Predator, ci toate filmele care au bugete în ambele grafuri; compararea bugetelor poate fi făcută de utilizator, sau prin includerea unui filtru care să păstreze doar filmele la care se găsesc bugete diferite:

```
select distinct ?x ?z ?c
where
{
graph :mynewmoviegraph {?x :hasBudget ?z}
graph :mymoviegraph {?x :hasBudget ?c}
filter (?z!=?c)
}
```

Acesta e un exemplu de detectare a contradicţiilor prin reguli încorporate în interogări ("dacă x are două bugete, afirmaţiile despre x sunt contradictorii").

Executați câteva interogări SPARQL cu metadate

Pentru următorul exemplu creați un al treilea fișier numit *metacontext.trig*.

```
@prefix : <http://expl.at#>.
:metadata
{
:mymoviegraph       :description "it should contain valid information";
                    :reputation 10;
                    :datasource  <http://www.imdb.com>.
:mynewmoviegraph :description "not so sure about this information";
                    :reputation 7;
:datasource :Robert.
}
```

Observați cum cele două grafuri anterior folosite pot fi la rândul lor descrise (sunt subiecte)! În general grafurile se descriu pentru a oferi clienţilor informaţii legate de provenienţa sau calitatea (reputaţia) lor. Astfel de descrieri de grafuri se mai numesc **metadate** și pot fi izolate în propriul lor graf, eventual administrat de cu totul altcineva decât cele descrise, pentru a asigura independenţa.
Acum putem să filtrăm rezultatele după reputația fiecărui graf:

Să se afișeze toate informațiile disponibile despre James Cameron, însă doar din graful cu reputația 10:

```
select *
where
{
graph :metadata {?x :reputation 10}
graph ?x {:JamesCameron ?b ?c}
}
```

Aceleași rezultate pot fi obținute dacă realizăm un filtru peste grafuri:

```
select ?x ?b ?c
where
{
graph :metadata {?x :reputation ?z}
graph ?x {?a ?b ?c}
filter ((?a=:JamesCameron)&&(?z=10))
}
```

4.2 Interogări SPARQL complexe

Următoarele exerciţii se vor aplica pe aceeaşi bază de cunoştinţe, în care s-au încărcat cele două fișiere:

Movies.trig:

```
@prefix : <http://expl.at#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
:mymoviegraph
{:JamesCameron      :directorOf :Avatar,:Terminator;
                    :hasName "James Cameron".
:JohnMcT            :directorOf :Predator, :DieHard;
                    :hasName "John McTiernan";
                    :birthInfo _:birthdetails.
_:birthdetails      :date "1951-01-08"^^xsd:date;
                    :place  :SUA.
:McG                :directorOf  _:somemovie;
                    :hasName "Joseph McGinty";
                    :hasNickname "McG".
:SamWorth           :hasName "Sam Worthington";
                   :playedIn :Rogue, _:somemovie;
                   :playedTheRole [:character :JakeSully;
                                   :movie :Avatar].
```

```
:Arnold          :playedIn :Terminator,:Predator;
                 :wasGovernorOf :California;
                 :hasName "Arnold Schwarzenegger";
                 :birthInfo [:date "1947-07-10"^^xsd:date;
                             :place  :Austria].
:LindaH          :hasName "Linda Hamilton";
                 :playedTheRole [:character :SarahConnor;
                             :movie :Terminator].
:ShBlack         :playedIn :Predator;
                 :hasName "Shane Black".
:Terminator      :hasTitle "Terminator";
                 :hasBudget 6400000 .
:Rogue           :hasTitle "Rogue".
:Avatar          :hasTitle "Avatar";
                 :hasBudget 237000000 .
:DieHard         :hasTitle "Die Hard".
:Predator        :hasTitle "Predator";
                 :hasBudget 18000000 .
:JakeSully       :hasName "Jake Sully".
:SarahConnor     :hasName "Sarah Connor".
:California      :hasName "California"@en, "Californie"@fr.
:Austria         :hasName "Austria"@en, "Autriche"@fr.
:SUA             :hasName "United States of America"@en, "Etats Unis d'Amerique"@fr.}
```

Movies2.trig:

```
@prefix : <http://expl.at#>.
:mynewmoviegraph
{:Titanic        :directedBy     :JamesCameron;
                 :hasActor       :LeoDiCaprio.
:Godzilla        :directedBy     :REmmerich.
:DieHard         :hasActor       :BruceWillis.
:JamesCameron    :directorOf     :Terminator.
:Predator        :hasBudget      100 .
:Titanic         :hasBudget      200 .
:DieHard         :hasBudget      300 .
:Godzilla        :hasBudget      400 .
_:somemovie      :hasBudget      500 .
}
```

La acestea mai adăugaţi încă un fişier (various.trig) în aceeaşi bază de cunoştinţe:

```
@prefix : <http://expl.at#>.
:Partonomy
{
:Computer :hasParts :Monitor, :CentralUnit, :Mouse, :Keyboard.
:Monitor :hasParts :Screen, :Frame, :Connectors, :Buttons.
:CentralUnit :hasParts :Case, :Motherboard, :CPU, :Disks, :PowerSupply.
:Keyboard :hasParts :Keys, :Circuits, :Cables.
:Motherboad :hasParts :Chipset, :Circuits, :Slots, :Board.
}
:SocialRelations
{
:Ana :hasFriend :Andrei, :Alin, :Andreea.
:Alin :hasFriend :Marian, :Valentin.
:Valentin :hasFriend :Marian, :Maria.
}
```

Ștergerea de cunoştinţe în interfaţa RDF4J

Să se șteargă din graful *mynewmoviegraph* afirmațiile ce au ca subiect pe James Cameron:
Alegeți *Modify-Remove*.
Tastați în *Context <http://expl.at#mynewmoviegraph>*
Tastați în *Subject <http://expl.at#JamesCameron>*

Să se șteargă din graful *mynewmoviegraph* afirmațiile despre nodul anonim _*:somemovie*. Nu putem folosi nodul anonim datorită instabilităţii identificatorilor anonimi, dar putem folosi bugetul, fiind singurul care are valoarea de 500.

Tastați în Context *<http://expl.at#mynewmoviegraph>*
Tastați în Object *"500"^^xsd:integer*

Ulterior vom studia cum putem realiza astfel de ştergeri prin interogări SPARQL de scriere. În continuare vom studia interogări SELECT complexe, dar vom presupune că cele două ştergeri din această secţiune au fost efectuate.

Interogări cu şabloane negate

Să se afișeze regizorii pentru Terminator și Predator[43]:

```
select ?x ?y
where
{
?x :directorOf ?y
filter (?y in (:Terminator,:Predator))
}
```

(operatorul IN poate fi de asemenea înlocuit cu disjuncție de condiții de filtrare multiple)

Cum se pot extrage regizorii care nu au lucrat la aceste filme? Aţi putea fi tentaţi să negați filtrarea:

```
select distinct ?x ?y
where
{
?x :directorOf ?y
filter (?y not in (:Terminator,:Predator))
}
```

Dar aceasta va afișa toți regizorii care au lucrat (şi) la ALTE filme! Rezultatele vor include aceiași indivizi, pentru că aceştia mai au şi ALTE filme în baza de cunoștințe.

Putem încerca să negăm proprietatea:

```
select distinct ?x ?y
where
{
?x !:directorOf ?y
filter (?y in (:Terminator,:Predator))
}
```

Dar acum obținem toate persoanele care au ALTĂ relație decât *directorOf* cu aceste două filme.

Pentru a obține soluția dorită avem nevoie de o subinterogare pentru a obţine rezultatul în mai mulţi paşi:

- în primul rând vom identifica pe aceia care au lucrat la cele două filme,
- apoi identificăm toți regizorii
- în final aplicăm o scădere între mulţimi: vom extrage primul set de soluții din al doilea set:

```
select ?x ?y
where
{
?x :directorOf ?y
filter (not exists
      {
      ?x :directorOf ?z
      filter (?z in (:Predator,:Terminator))
      })
```

Pas2. Identifică toți regizorii (→ ?x :directorOf ?y)
Pas3. Elimină regizorii celor două filme din lista completă a regizorilor (→ filter (not exists)
Pas1. Identifică regizorii celor două filme (→ ?x :directorOf ?z)

[43] Documentul nu va indica prefixele la fiecare interogare, dar ele trebuie trecute (în versiuni recente se inserează automat la prima tastare a prefixului)

```
}
```

Observați variabila comună ?x – în acest fel ne asigurăm că rezultatele subinterogării vor fi căutate în rezultatele interogării principale și eliminate. Singurul rezultat acum ar trebui să fie regizorul McG.

Același rezultat se poate obţine şi folosind clauza MINUS, care sugerează mai clar faptul că avem de a face cu o "scădere" între două mulţimi de soluţii:

```
select ?x ?y
where
{
?x :directorOf ?y
minus {
    ?x :directorOf ?z
    filter (?z in (:Predator,:Terminator))
    }
}
```

Același rezultat poate fi obținut şi printr-o a treia metodă:

```
 select ?x ?y
where
{
?x :directorOf ?y
optional
        {
        ?x :directorOf ?z.
        filter (?z in (:Predator,:Terminator))
        }
filter (!bound(?z))
}
```

Pas1. Identifică toți regizorii şi filmele lor

Pas2. Ataşează fiecărei soluţii filmele Terminator sau Predator DOAR DACĂ regizorii au lucrat la acestea. La ceilalţi regizori nu ataşa nimic.

Pas3. Păstrează doar soluţiile pentru care nu s-a ataşat nimic la Pasul 2!

Negarea clauzei BOUND va păstra doar acele rezultate din interogarea principală pentru care subinterogarea OPTIONAL nu a găsit o conexiune cu Terminator/Predator. Clauza BOUND este de obicei folosită în combinație cu clauza FILTER pentru a verifica dacă o subinterogare a avut rezultate sau nu. Această variantă, deşi pare mai complicată, poate fi necesară în sistemele care suportă versiunea SPARQL 1.0 (clauzele MINUS şi NOT EXISTS au fost introduse abia în SPARQL 1.1)

Deşi par să funcţioneze identic, există anumite diferențe între clauzele MINUS și FILTER NOT EXISTS. Următoarele două interogări nu vor da aceleași rezultate:

```
select ?x ?y
where
{
?x :directorOf ?y.
minus {?z :directorOf :Predator}
}
```

(interogarea principală şi subinterogarea se execută separat, apoi are loc "scăderea" dintre mulţimi, însă aceasta se face pe baza conexiunilor dintre şabloane (variabile comune); cum nu există nicio variabilă comună pentru a conecta rezultatele, şablonul MINUS nu va elimina nimic şi rămânem cu toţi regizorii).

```
select ?x ?y
where
{
?x :directorOf ?y.
filter not exists {?z :directorOf :Predator}
}
```

(aici interogarea principală găsește primul rezultat, apoi testează filtrul; deoarece există un regizor pentru Predator, filtrul returnează FALSE şi rezultatul nu e păstrat; apoi se găsește la al doilea rezultat ș.a.m.d. ... în final nu rămânem cu nicio soluţie, deoarece filtrul negat returnează FALSE de fiecare dată)

Concluzie:

- **clauza MINUS selectează toate rezultatele pentru ambele interogări şi abia la sfârşit aplică eliminarea**

- **clauza FILTER testează filtrul după fiecare fiecare rezultat şi decide dacă să îl elimine sau nu în funcţie de ce returnează filtrul.**

O altă diferență este evidențiată de exemplele următoare, care din nou dau rezultate diferite:

```
select ?x ?y
where
{
?x :directorOf ?y.
minus
{
?z :directorOf :Predator
filter (?z=?x)
}
}
```

- Regizorul filmului Predator nu este eliminat din solutie, deoarece subinterogarea din MINUS se execută separat de interogarea principală şi nu are acces la valoarea lui x!

```
select ?x ?y
where
{
?x :directorOf ?y.
filter not exists
{
?z :directorOf :Predator
filter (?z=?x)
}
}
```

- Regizorul filmului Predator este eliminat din soluţie! Pentru fiecare rezultat al interogării principale, variabila ?x a fost pasată spre subinterogare permiţând aplicarea filtrului

Interogări cu şabloane complexe

Următoarele exemple vor fi executate pe graful mynewmoviegraph, care ar trebui să conțină următoarele afirmații în care relaţia de regizor are acum o singură formă - directedBy (după ștergerile realizate la început):

```
@prefix : <http://expl.at#>.
:mynewmoviegraph
{:Titanic      :directedBy     :JamesCameron;
               :hasActor       :LeoDiCaprio;
               :hasBudget      200 .
:Godzilla      :directedBy     :REmmerich;
               :hasBudget      400 .
:DieHard       :hasActor       :BruceWillis;
               :hasBudget      300 .
:Predator      :hasBudget      100 .}
```

Să se găsească toate filmele care au regizori SAU actori declaraţi:

```
select *
from :mynewmoviegraph
where
{
?m1 :directedBy ?dir.
?m2 :hasActor ?act
}
```

A se observa modul neplăcut de afişare a rezultatelor. S-au găsit regizorii, s-au găsit actorii, apoi s-a realizat un produs cartezian între soluţii (toate combinaţiile posibile între soluţiile celor două şabloane, pe coloane diferite). Acest lucru este cauzat de folosirea unor variabile diferite – tabelul de afişare a rezultatelor va construi o coloană pentru fiecare variabilă.

Rezultatele se pot oferi într-un mod mai uşor de citit, dacă folosim o singură variabilă pentru filme şi aplicăm o reuniune:

```
select *
```

```
from :mynewmoviegraph
where
{
{?mov :directedBy ?dir}
union
{?mov :hasActor ?act}
}
```

Folosind UNION, fiecare șablon va fi tratat ca o interogare separată, apoi rezultatele sunt reunite şi aranjate în coloane corespunzătoare variabilelor. În consecinţă filmele apar într-o singură coloană, iar regizorii și actorii pe coloane diferite.

Să se extragă toate filmele care au declarați atât regizor cât ŞI actori:

```
select *
from :mynewmoviegraph
where
{
?mov :directedBy ?regizor.
?mov :hasActor ?actor
}
```

A se observa că în absenţa lui UNION variabila comună conectează cele două șabloane şi va obţine doar filmul ce corespunde ambelor şabloane. Cu alte cuvinte, acest gen de utilizare corespunde unui ŞI logic, în timp ce prezenţa lui UNION corespunde unui SAU logic.

Să se găsească toate filmele cu regizori. Dacă se găsesc, să li se obţină și actorii:

```
select *
from :mynewmoviegraph
where
{
?mov :directedBy ?dir
optional {?mov :hasActor ?act}
}
```

Clauza OPTIONAL va realiza o conectare opţională a şabloanelor – la filmele care au actori, aceştia vor fi afişaţi alături de regizor, la filmele care nu au actori declaraţi, doar regizorul va fi vizibil. Atenţie la diferenţa faţă de cazul precedent, în care apar doar filmele pentru care s-au declarat atât regizor cât şi actor.

Clauza OPTIONAL este de obicei folosită pentru explorare, pentru a solicita proprietăți care s-ar putea să lipsească pentru unele subiecte.

Să se găsească toate filmele care au declarați actori dar NU și regizori:

```
select *
from :mynewmoviegraph
where
{
?mov :hasActor ?act
minus {?mov :directedBy ?dir}
}
```

Avem de a face cu negaţia (diferenţa între mulţimi) despre care am discutat deja.

Să se găsească toate filmele pentru care nu avem NICI actori NICI regizori:

```
select *
from :mynewmoviegraph
where
{
?mov :hasBudget ?b
minus
{{?mov :directedBy ?x} union {?mov :hasActor ?x}}
}
```

Aici am îmbinat un SAU logic cu negaţia şabloanelor. Mai întâi filmele au fost detectate după prezenţa bugetului, având în vedere că e singura proprietate care apare la toate filmele, deci poate fi considerată "definitorie" pentru ca ceva să poată fi considerat "film". Apoi, am eliminat soluţiile care au declaraţi regizori sau actori.

Rezultate similare se obţin combinând clauzele OPTIONAL și BOUND:

```
select *
from :mynewmoviegraph
where
{
?mov :hasBudget ?b
optional {?mov :directedBy ?dir}
optional {?mov :hasActor ?act}
filter (!bound(?dir)&&!bound(?act))
}
```

Interpretare:

- din nou, am folosit bugetul ca şi criteriu definitoriu pentru a identifica filmele;
- apoi cu OPTIONAL am căutat posibile conexiuni cu regizori şi actori
- apoi cu FILTER am păstrat doar soluţiile la care nu am găsit astfel de conexiuni

Să se găsească toate filmele care au declaraţi fie actori fie regizori, dar nu ambele:

```
select *
from :mynewmoviegraph
where
{
{?mov :hasActor ?act
minus {?mov :directedBy ?dir}}
union
{?mov :directedBy ?dir
minus {?mov :hasActor ?act}}
}
```

Rezultate similare s-ar putea obține prin filtrare:

```
select ?mov
from :mynewmoviegraph
where
{
?mov :hasBudget ?b
filter (
((exists {?mov :hasActor ?act})&&(not exists {?mov :directedBy ?dir}))
||
((exists {?mov :directedBy ?dir})&&(not exists {?mov :hasActor ?act}))
)
}
```

Să se găsească filmele care nu au declarați atât actorii cât și regizorii (fie doar una din variante, fie niciuna):

```
select ?mov
from :mynewmoviegraph
where
{
?mov :hasBudget ?b
minus
{
?mov :directedBy ?dir.
?mov :hasActor ?act
}
}
```

Să se găsească filmele care au actori, regizori sau ambii, dar nu și acelea care nu au nimic din acestea.

```
select *
from :mynewmoviegraph
where
{
```

```
?mov :hasBudget ?b
optional {?mov :directedBy ?dir}
optional {?mov :hasActor ?act}
filter (bound(?dir)||bound(?act))
}
```

Dacă vrem să obţinem doar lista cu filme din ultimul exemplu, putem folosi și următoarea interogare:

```
select ?mov
from :mynewmoviegraph
where
{
?mov :hasBudget ?b
filter
((exists {?mov :directedBy ?dir})||(exists{?mov :hasActor ?act}))
}
```

Să se găsească filmul cu cel mai mare buget, dintre filmele care au declarat regizorul:

```
select ?x ?maxb
from :mynewmoviegraph
where
{
?x :hasBudget ?maxb
    {
    select  (max(?b) as ?maxb)
    where
    {?mov :directedBy ?a.?mov :hasBudget ?b}
    }
}
```

Să se construiască propoziții precum "x had a budget of y" pentru fiecare film în cadrul tuturor grafurilor. Dacă bugetul nu este disponibil, să se afișeze propoziția "x has no budget info available":

```
select (if(bound(?b),concat(?t," had a budget of ",str(?b)),concat(?t," has no budget info available")) as ?sentence)
where
{
?x :hasTitle ?t
optional {?x :hasBudget ?b}
}
```

Interogări cu proprietăți înlănțuite

Din graful *mymoviegraph*, să se extragă bugetele filmelor care au fost regizate de regizorul filmului Avatar.

```
select ?b
from :mymoviegraph
where
{
?dir :directorOf :Avatar.
?dir :directorOf ?mov.
?mov :hasBudget ?b
}
```

Deoarece ne interesează doar bugetul, variabilele intermediare pot fi eliminate prin crearea unei căi de proprietăți înlănțuite:

```
select ?b
from :mymoviegraph
where
{
:Avatar ^:directorOf/:directorOf/:hasBudget ?b
}
```

Explicații:

- Caracterul ”/” separă diverși pași (afirmații) ai înlănțuirii
- Caracterul ”^” indică faptul că relaţia *directorOf* trebuie traversat în direcție inversă (de la obiect la subiect) Astfel calea se traduce ”de la Avatar, mergi înapoi pe relația *directorOf* (pentru a

determina regizorul), apoi de acolo mergi înainte pe toate relațiile *directorOf* (pentru a determina toate filmele acelui regizor), apoi mai departe pe relațiile *hasBudget* (pentru a obţine bugetele)”

Să se găsească actorii din graful *mymoviegraph* împreună cu datele de naștere, dacă sunt disponibile:

```
select distinct ?act ?d
from :mymoviegraph
where
{
{?act :playedIn ?x}
union
{?act :playedTheRole ?y}
optional {?act :birthInfo ?bi.?bi :date ?d}
}
```

A se observa că actorii pot fi obținuți după implicarea acestora în relația *playedIn* sau în relația *playedTheRole*. Apoi, clauza OPTIONAL va verifica disponibilitatea informațiilor legate de data de naștere (a nodurilor anonime) și va extrage data.

Varianta cu proprietăți înlănțuite ar fi:

```
select distinct ?d
from :mymoviegraph
where
{
?x (^:playedIn|^:playedTheRole)/:birthInfo/:date ?d
}
```

Calea se traduce după cum urmează: ”de la resursa arbitrară ?x se merge înapoi fie pe relația *playedIn*, fie pe relația *playedTheRole* (a se observa caracterul "|" pentru SAU - o înlănţuire cu alternative), găsind astfel actorii; apoi de la aceștia se merge înainte pe relația *birthInfo* (pentru a găsi nodurile anonime) și mai departe pe relația *date* (pentru a găsi data nașterii).

Să se găsească actorii care au jucat în filmele regizorului lui Terminator, iar dacă sunt disponibile, să se obţină și datele lor de naștere:

```
select distinct ?act ?d
from :mymoviegraph
where
{
?dir :directorOf :Terminator.
?dir :directorOf ?mov
{?act :playedIn ?mov}
union
{?act :playedTheRole ?bn1. ?bn1 :movie  ?mov}
optional {?act :birthInfo ?bn2. ?bn2 :date ?d}
}
```

Varianta cu proprietăți înlănțuite (care ne dă doar datele de naştere) ar fi:

```
select distinct ?d
from :mymoviegraph
where
{
:Terminator ^:directorOf/:directorOf/(^:playedIn|(^:movie/^:playedTheRole))/:birthInfo/:date ?d
}
```

Calea se traduce în felul următor: ”de la Terminator se merge înapoi pe relația *directorOf* (pentru a găsi regizorul acestuia), apoi se merge înainte pe aceeași relație (pentru a găsi toate filmele acestuia), apoi se merge înapoi fie pe relația *playedIn*, fie pe succesiunea *movie/playedTheRole* (pentru a găsi toți actorii din aceste filme), de acolo se merge pe relația *birthInfo* (pentru a găsi nodurile anonime) și mai departe pe relația *date* (pentru a extrage datele de naștere)”.

Observaţi totuşi că varianta cu înlănţuire obţine doar data naşterii, deci informaţia de la capătul lanţului. În general în interogări înlănţuite nodurile intermediare se pierd (fiind înlocuite cu "/") şi se returnează doar variabilele de la capetele lanţului.

Să se extragă toți colaboratorii lui Arnold (care au jucat în același film ca și el sau care au regizat filmele în care apare):

```
select distinct *
from :mymoviegraph
where
{
:Arnold (:playedIn/^:playedIn)|(:playedIn/^:directorOf)|(:playedIn/^:movie/^:playedTheRole) ?collab
filter (?collab!=:Arnold)
}
```

Pornind de la Arnold și relația acestuia *playedIn* (care ne dă filmele în care a jucat), se iau trei căi alternative în direcția opusă (date de semnul "^"): combinația dintre *playedIn* și *movie-playedtheRole* ne va da actorii din aceste filme; relația *directorOf* ne va da regizorii. În final, Arnold este scos din lista de rezultate prin filtrare (deoarece va fi găsit ca actor al filmelor proprii, prin parcurgerea primei căi).

Să se obțină informațiile conectate la actori, cu excepția numelor acestora:

```
select distinct ?info
from :mymoviegraph
where
{
?x (^:playedIn|^:playedTheRole)/!:hasName ?info
}
```

A se observa folosirea semnului "!" ca negaţie, pentru a elimina din rezultate o anumită proprietate.

Să se extragă toate lucrurile cu care Arnold are alte relaţii decât hasName sau birthInfo:

```
select *
from :mymoviegraph
where
{
:Arnold !(:hasName|:birthInfo) ?info
}
```

Să se extragă toate filmele și bugetele la care a lucrat JohnMcT:

```
select distinct ?info
from :mymoviegraph
where
{
:JohnMcT :directorOf/:hasBudget? ?info
}
```

A se observa folosirea semnului "?" pentru a indica un pas opțional în cadrul înlănțuirii. Cu alte cuvinte, interogarea va căuta mai întâi doar obiectele relației *directorOf* (deci filmele), apoi combinațiile date de *directorOf/hasBuget* (bugetele filmelor) și va stoca toate rezultatele pe o singură coloană, dată de variabila ?*info*. Caracterul opţional aplicat unora din paşii înlănţuirii permit aşadar obţinerea unor informaţii intermediare din cadrul lanţului.

Comparați rezultatele obținute cu următoarele, care extrag aceeași informații dar folosesc variabile separate pentru a colecta filmele și bugetele. Acest lucru e posibil datorită utilizării variabilei intermediare ?mov pentru filme (care nu poate apare în înlănţuire).

```
select *
from :mymoviegraph
where
{
:JohnMcT :directorOf ?mov
optional {?mov :hasBudget ?b}
}
```

Reamintim că la început am adăugat un graf *Partonomy* ce descrie modul de descompunere a unui calculator în componente. Să se extragă din graful *Partonomy* toate părțile unui computer, în mod recursiv (deci şi părţile părţilor acestora):

```
select *
from :Partonomy
where
{
:Computer :hasParts+ ?x
}
```

Interogarea e executată în felul următor: se caută obiectele relației *hasParts*, apoi ale lanţului *hasParts/hasParts*, apoi *hasParts/hasParts/hasParts* etc. până când nu mai sunt găsite rezultate. A se observa semnul "+" pentru a indica faptul că aceeaşi relaţie se înlănţuie cu ea însăşi de un număr arbitrar de ori.

Să se extragă toate componentele de pe același nivel cu *Motherboard* în ierarhia *Partonomy*:

```
select *
from :Partonomy
where
{
:Motherboard ^:hasParts/:hasParts ?x
}
```

Reamintim că la început am adăugat un graf *SocialRelations* pentru a descrie o reţea socială. Să se extragă toți prietenii lui Alin:

```
select *
from :SocialRelations
where
{
:Alin :hasFriend|^:hasFriend ?x
}
```

Deoarece relația de prietenie funcționează în ambele sensuri, trebuie să luăm în considerare atât relațiile *hasFriend* care îl au pe Alin ca subiect, cât şi pe cele care îl au drept obiect. Cu ajutorul acestei interogări găsim pe toţi cei conectaţi cu Alin în mod direct. Aceasta e o interogare uzuală în grafuri ce trebuie interpretate ca reţele bidirecţionale, în care relaţiile trebuie interpretate ca fiind reciproce!

Să se calculeze numărul de prieteni pentru fiecare membru din graful *SocialRelations*:

```
select ?x (count(?y) as ?FriendCount)
from :SocialRelations
where
{
?x :hasFriend|^:hasFriend ?y
}
group by ?x
```

Aici am extins interogarea de mai sus pentru a număra conexiunile fiecărui individ.

Interogări ASK și DESCRIBE

Interogările ASK nu returnează date ci răspunsuri sub forma valorilor booleene(Yes sau No):

James Cameron a regizat Avatar? (Da)

```
ask
{:JamesCameron :directorOf :Avatar}
```

James Cameron a regizat Predator? (Nu)

```
ask
{:JamesCameron :directorOf :Predator}
```

James Cameron a regizat ceva al cărui titlu se termină cu ”or”? (Da)

```
ask
{
```

```
:JamesCameron :directorOf/:hasTitle ?t
filter regex(?t,"or$","i")
}
```

James Cameron a regizat vreun film pentru care nu avem informații disponibile în ceea ce privește bugetul? (Nu)

```
ask
{
:JamesCameron :directorOf ?mov
minus {?mov :hasBudget ?b}
}
```

James Cameron a regizat vreun film în care a jucat Arnold? (Da)

```
ask
{
:JamesCameron :directorOf/^:playedIn :Arnold
}
```

Interogările DESCRIBE returnează toate afirmaţiile care includ un anumit termen de tip URI sau sunt conectate la acesta prin noduri anonime. Să se găsească toate cunoștințele pe care le avem despre Terminator:

```
describe :Terminator
```

Observaţi că printre rezultate se găseşte şi o afirmaţie care nu îl include pe Terminator, dar are un nod anonim conectat la Terminator.

Să se găsească toate cunoștințele despre toate filmele regizate de James Cameron:

```
describe *
{
:JamesCameron :directorOf ?mov
}
```

Reguli încorporate în interogări

Problema cu care ne confruntăm la baza de cunoştinţe cu filme este faptul că există mai multe proprietăţi definitorii pentru calitatea de "actor". Pentru a extrage toți actorii, trebuie să luăm în considerare două relații: *playedIn* și *playedTheRole*:

```
select distinct ?act
{
{?act :playedIn ?mov}
union
{?act :playedTheRole ?r}

}
```

În unele interogări a trebuit să luăm în considerare aceste variante, ceea ce complică interogările şi poate crea probleme pentru clienţii externi. Un client care nu este familiarizat cu afirmaţiile stocate nu ar avea cum să știe că trebuie să ţină cont de ambele proprietăţi!

Putem rezolva această problemă prin uniformizarea modului de detectare a actorilor. Cea mai directă cale de a obţine acest lucru este să adăugăm la baza de cunoştinţe afirmaţii de forma:

```
Arnold a :Actor.
```

Putem genera astfel de afirmații prin definirea regulii următoare: "Dacă X este subiect implicat în relații precum *playedIn* sau *playedTheRole,* atunci X este actor". Astfel de reguli se pot încorpora în interogări de tip CONSTRUCT sau INSERT. În această secţiune vom exemplifica doar CONSTRUCT, iar în secţiunea următoare INSERT. Principala diferenţă între acestea este:

- CONSTRUCT generează afirmaţii noi la client (pe ecran, în cazul nostru), fără a le salva în baza de cunoştinţe. Salvarea lor trebuie făcută de client, dacă acesta e mulţumit de rezultat;
- INSERT este o interogare de scriere, asigurând şi salvarea afirmaţiilor generate în baza de cunoştinţe.

Exemplu:
```
construct
{?act a :Actor}
where
{
?act :playedIn|:playedTheRole ?x
}
```

Remarcaţi relaţiile generate de forma "x este Actor", afişate pe ecran. Salvarea lor trebuie făcută manual:
- Folosim butonul *Download* din ecranul cu rezultate pentru a le salva într-un fișier (selectați formatul TriG pentru download)
- Deschidem fișierul și copiem conținutul acestuia (cu Copy)
- Alegem opțiunea *Add* (în RDF4J)
- Introduceți (cu Paste) conținutul fișierului descărcat în secțiunea *RDF Content*
- Selectați formatul pentru sintaxa introdusă – TriG (la *Data format*)

Acum noile afirmaţii au fost adăugate, iar lista cu actorii este disponibilă prin interogări mult mai simple decât înainte:
```
select ?x
where {?x a :Actor}
```

În acest moment, baza noastră de cunoştinţe include un element aparte pe care nu îl avea până acum: o **clasă** (mulţime). Prin generarea relaţiilor de **apartenenţă la clasă** (verbul "a fi" la singular), putem foarte uşor să obţinem lista actorilor, dar şi să punem una din întrebările cele mai importante în bazele de cunoştinţe: **Ce este X?** Această întrebare are forma:

```
select ?x
where {:Arnold a ?x}
```

În mod ideal, pentru orice X din baza de cunoştinţe trebuie ca RDF4J să poată răspunde la întrebarea **Ce este X?** Cu alte cuvinte, orice URI trebuie să aparţină unei mulţimi. În cazul nostru, la această întrebare se poate răspunde doar în cazul actorilor. Aceeaşi tehnică se poate aplica şi pentru regizori ("Dacă X e subiect al relației directorOf, X este regizor"), pentru filme ("Dacă X are titlu, buget sau este obiect în relațiile directorOf sau playedIn, atunci X este film"), pentru locuri geografice ("Dacă X este obiect al înlănțuirii birthInfo/place, atunci X este locaţie").

În continuare, să presupunem că dorim să obținem lista cu actorii și filmele în care au jucat. Din nou suntem nevoiţi să folosim ambele relaţii, pentru a detecta toate conexiunile posibile între actori şi filme (clasa Actor nu este de folos pentru a ajunge la filme!)
```
select distinct ?act ?mov
{
{?act :playedIn ?mov}
union
{?act :playedTheRole/:movie ?mov}
}
```

Acum vom aplica o uniformizare a relațiilor, prin generarea unei relații noi din cele două relații cu semnificaţie similară:
```
construct {?act :isActorIn ?mov}
where
{
{?act :playedIn|(:playedTheRole/:movie) ?mov}
}
```

Precum mai înainte, descărcaţi manual rezultatul şi reîncărcaţi-l în RDF4J pentru a fi disponibil viitoarelor interogări.

Acum putem să obținem lista actorilor și a filmelor acestora mult mai simplu decât înainte:

```
select ?act ?mov
where
{
?act :isActorIn ?mov
}
```

O relație de colaborare directă poate fi adăugată între regizorii și actorii care au lucrat împreună la același film. Pentru a evita încărcarea rezultatelor manual, de acum înainte vom folosi o interogare INSERT (trebuie să fie introdusă în ecranul *SPARQL Update* și nu *Query*).

```
insert {?x :workedWith ?y}
where {?x :directorOf/^:isActorIn ?y}
```

Salvarea manuală nu mai e necesară, iar colaborările pot fi extrase acum mai rapid:

```
select ?x ?y
where {?x :workedWith ?y}
```

În unele cazuri, este util să se folosească comanda CONSTRUCT pentru a extrage un subgraf din baza de cunoştinţe:

```
construct where {?x :hasName ?y}
```

Deoarece din comandă lipsește primul șablon, rezultatele de la șablonul WHERE vor deveni chiar soluția oferită. Cu alte cuvinte generăm niște date care sunt deja în baza de cunoștințe. **Care este utilitatea acestei operaţii, având în vedere că am putea obţine un rezultat similar şi cu SELECT?**

Diferenţa este dată de sintaxa în care se oferă rezultatele! Verificaţi opţiunea *Download format*, cu care puteţi alege formatul în care să descărcaţi rezultatele. Beneficiul este că CONSTRUCT returnează rezultatele sub forma unui graf RDF, care ne este oferit în oricare din sintaxele RDF (TriG, TriX etc.). Asta înseamnă că poate fi uşor descărcat, transferat spre alte baze de cunoştinţe etc.

Verificați acum modul în care returnează rezultatele comanda SELECT:

```
select ?x ?y
where {?x :hasName ?y}
```

Opțiunile de download (*Download format*) sunt cu totul altele! Acestea sunt SPARQL/XML, SPARQL/JSON etc. – niciunul dintre ele nu este o sintaxă RDF, ci structuri de date XML, JSON, CSV etc. ce exprimă o structură tabelară cu înregistrări (un tabel cu o coloană pentru fiecare variabilă şi un rând pentru fiecare soluţie). Astfel de rezultate sunt menite să fie integrate în interfaţa cu utilizatorul (deci procesate prin tehnici XML/JSON) şi nu să fie reîncărcate într-o bază de cunoştinţe (cum este cazul CONSTRUCT).

4.3 Interogări SPARQL Update în RDF4J

Am văzut deja o interogare de scriere – INSERT. În această secţiune intrăm în mai multe detalii legate de operaţiile de scriere.

Creaţi o bază de cunoştinţe nouă, fără a folosi ecranul *Add*. De data aceasta vom realiza adăugări şi modificări de cunoştinţe prin interogări de scriere (numite şi interogări "SPARQL Update"). Acestea se realizează în ecranul *SPARQL Update*, cu următoarele consecinţe în privinţa modului de lucru:

- când trebuie să executaţi un SELECT pentru a vedea efectul unor modificări, trebuie să reveniţi în ecranul *Query*;
- ca alternativă, adesea veţi putea vedea mai rapid modificările în ecranele *Export* (pentru a vedea toată baza de cunoştinţe) sau *Contexts* (pentru a avea acces selectiv la grafurile pe care le creaţi);

Metoda 1: Adăugarea de afirmaţii din fişiere on-line (direct accesibile prin HTTP)

Pentru următoarele exemple creați un fișier nou numit *social.ttl* cu următorul conținut:

```
@prefix : <http://expl.at#>.
:Anna :hasFriend :Andrew, :Peter, :Mary.
:Andrew :hasFriend :George.
:George :hasFriend :Roxanne.
```

De această dată fișierul nu este în sintaxă TriG, deci nu precizează graful în care trebuie încărcate afirmaţiile. Vom specifica graful în interogare!

Faceţi acest fișier disponibil prin HTTP, prin plasarea sa în folderul rădăcină pentru pagini Web găzduite în Tomcat (Tomcat/webapps/Root). Putem verifica disponibilitatea acestuia în browser, la adresa: http://localhost:8080/social.ttl .

În RDF4J, pentru interogări de scriere (insert/update/delete) vom folosi ecranul *SPARQL Update* (sub *Modify*) în loc de *Query*!

Tastați următoarea interogare, incluzând şi prefixul (baza de cunoştinţe fiind goală, trebuie să includem manual prefixul, sau să îl definim în prealabil în ecranul *Namespaces* pentru a evita tastarea sa la fiecare interogare):

```
prefix : <http://expl.at#>
load <http://localhost:8080/social.ttl> into graph :FacebookRelations
```

V
erificați acum la *Contexts* graful nou adăugat.

Metoda 2: Adăugarea de afirmaţii specificate direct în interogare

Să se tasteze în ecranul *SPARQL Update* următoarea comandă, care creează 2 grafuri noi indicate explicit în cadrul interogării, împreună cu conţinutul lor (de aici încolo prefixul ar putea să apară automat, deoarece a fost detectat în fişierul încărcat cu LOAD în exerciţiul anterior; dacă nu apare automat în ecranul de interogare, prefixul trebuie inclus manual ca până acum![44]):

```
insert data
{
graph :FamilyRelations
        {
        :MarySmith :isRelativeOf :JohnSmith.
        :JohnSmith :isRelativeOf :AnnaSmith,:GeorgeParker.
        }
graph :OtherData
        {
        :EllenSmith :hasHairColor :Red; :hasAge 20; :daughterOf :JackSmith.
        :MarySmith :daughterOf :JackSmith.
        }
}
```

Să se creeze relația *isRelative* în cadrul grafului *FamilyRelations* pentru fiecare relație *daughter* găsită în graful *OtherData*:

```
insert
        {
        graph :FamilyRelations {?x :isRelativeOf ?y}
        }
where
        {
        graph :OtherData {?x :daughterOf ?y}
        }
```

[44] Prefixul nu va fi inclus în fiecare exemplu, dar asta nu înseamnă că el nu trebuie să apară!

Verificați conținutul grafului *FamilyRelations* pentru a vedea că au apărit relaţii *isRelative* între EllenSmith şi JackSmith, respectiv între MarySmith şi JackSmith. Puteţi face această verificare:

- Printr-o interogare SELECT (atenţie, pentru interogări select trebuie să treceţi în ecranul *Query*, apoi pentru interogări de scriere reveniţi în ecranul *SPARQL Update*):

```
select *
from :FamilyRelations
where {?x ?y ?z}
```

- Cu un click pe opţiunea *Contexts*, apoi alegeţi graful *FamilyRelations* pentru a-i vedea tot conţinutul

De aici încolo pentru oricare din operaţiile de scriere verificaţi pe una din aceste căi succesul operaţiei.

Să se creeze în graful *OtherData* o relație *fatherOf* pentru fiecare relație *daughterOf*, dar cu direcţia inversată (tatăl să apară ca subiect):

```
with :OtherData
insert
{?y :fatherOf ?x}
where
{?x :daughterOf ?y}
```

Verificați conținutul grafului *OtherData* pentru a vedea rezultatele.

Important: Acesta este un exemplu de **REGULĂ încorporată în interogări** ("dacă x e fiica lui y, y e tatăl lui x", respectiv ceva mai înainte am avut "dacă x e fiica lui y, x e rudă cu y").
Reamintim că regulile au rolul de a executa deducţii logice, generând afirmaţii noi ("concluzii") din cele existente. *Regulile pot fi prezente fie în interogări* (cazul de faţă), fie *sub formă de axiome* (vor fi discutate ulterior).

- **Dacă regulile sunt prezente în interogări**, generarea concluziilor are loc prin executarea interogării (cazul de faţă). Pentru această tehnică se folosesc:
    - interogările CONSTRUCT (caz în care afirmaţiile generate sunt oferite clientului; acesta trebuie să le salveze în baza de cunoştinţe)
    - interogările INSERT (care se ocupă de salvarea afirmaţiilor generate, dar nu le oferă clientului; acesta trebuie să le obţină ulterior cu SELECT);
- **Dacă regulile sunt prezente ca axiome**, generarea concluziilor are loc fără intervenţie umană (după cum se va vedea ulterior). Pentru această tehnică se folosesc terminologiile standardizate şi motoarele inferenţiale.

Unirea, copierea și redenumirea grafurilor

Să se adauge conținutul grafului *FamilyRelations* în graful *OtherData*:

```
add graph :FamilyRelations to graph :OtherData
```

Verificați conținutul grafului *OtherData* pentru a observa rezultatele combinate. ADD a avut ca efect o uniune a conţinutului vechi cu cel nou în graful destinaţie.

Înlocuiți conținutul grafului *OtherData* cu conținutul grafului *FacebookRelations*:

```
copy graph :FacebookRelations to graph :OtherData
```

Verificați conținutul grafului *OtherData* pentru a observa înlocuirea. Spre deosebire de ADD, COPY va distruge conținutul existent în graful destinație!

Mutați conținutul grafului *OtherData* în **graful implicit** (default):

```
move graph :OtherData to default
```

Observați în ecranul *Contexts* că nu mai există graful *OtherData*! Totuşi afirmaţiile din acesta nu s-au pierdut. Puteți verifica prin opțiunea *Export* că ele au rămas în baza de cunoştinţe dar nu mai au context (nu mai aparţin unui graf identificat)! Mai exact, ele aparţin acum "grafului implicit". **Graful implicit** (default)

este un graf fără identificator, care există în RDF4J în mod implicit atunci când nu suntem interesaţi să lucrăm cu grafuri (de exemplu când facem upload de fişiere Turtle în loc de TriG).

Se poate considera şi că MOVE funcționează ca o "redenumire" – în acest caz, graful şi-a schimbat identificatorul din *OtherData* în nimic. În loc de cuvântul cheie DEFAULT se putea folosi un URI, caz în care devine mai clar faptul că MOVE e o operaţie de redenumire de grafuri (chiar dacă numele său ar putea sugera altceva).
Spre deosebire de COPY, cu MOVE se elimină atât graful sursă (în sensul că e "redenumit", păstrându-i-se însă conţinutul) cât şi conţinutul graful destinaţie (dacă exista deja un graf cu noul identificator, conţinutul său va fi înlocuit, ca la COPY).

Pentru a vedea doar afirmaţiile care şi-au pierdut contextul (graful) se poate accesa conținutul grafului implicit prin interogări cu clauza FROM DEFAULT:

```
select *
from default
where {?x ?y ?z}
```

Atunci când nu se folosește clauza FROM sau GRAPH, interogările se execută pe toată baza de cunoştinţe (pe uniunea tuturor grafurilor, inclusiv cel implicit):

```
select *
where {?x ?y ?z}
```

Să se mute conținutul grafului *FamilyRelations* în graful implicit:

```
move graph :FamilyRelations to default
```

Operaţia e similară cu cea de dinainte, dar acum s-a distrus graful *FamilyRelations*, mutându-i-se conţinutul în graful implicit. Cum MOVE distruge conţinutul grafului destinaţie, ceea ce s-a mutat mai înainte din *OtherData* s-a pierdut (verificați prin opțiunea *Export* că acum afirmațiile fără context sunt cele care au relația *isRelative*).
Acum am rămas doar cu graful *FacebookRelations* şi cu graful implicit (cu conţinutul preluat din fostul graf *FamilyRelations*).

Subject	Predicate	Object	Context
<http://expl.at#Anna>	<http://expl.at#hasFriend>	<http://expl.at#Andrew>	<http://expl.at#FacebookRelations>
<http://expl.at#Anna>	<http://expl.at#hasFriend>	<http://expl.at#Peter>	<http://expl.at#FacebookRelations>
<http://expl.at#Anna>	<http://expl.at#hasFriend>	<http://expl.at#Mary>	<http://expl.at#FacebookRelations>
<http://expl.at#Andrew>	<http://expl.at#hasFriend>	<http://expl.at#George>	<http://expl.at#FacebookRelations>
<http://expl.at#George>	<http://expl.at#hasFriend>	<http://expl.at#Roxanne>	<http://expl.at#FacebookRelations>
<http://expl.at#MarySmith>	<http://expl.at#isRelativeOf>	<http://expl.at#JohnSmith>	
<http://expl.at#MarySmith>	<http://expl.at#isRelativeOf>	<http://expl.at#JackSmith>	
<http://expl.at#JohnSmith>	<http://expl.at#isRelativeOf>	<http://expl.at#AnnaSmith>	
<http://expl.at#JohnSmith>	<http://expl.at#isRelativeOf>	<http://expl.at#GeorgeParker>	
<http://expl.at#EllenSmith>	<http://expl.at#isRelativeOf>	<http://expl.at#JackSmith>	

Figura 31 Conținutul bazei de cunoștințe vizibil la opțiunea Export

Ștergerea afirmaţiilor cu interogări SPARQL

Să se șteargă o afirmaţie indicată direct prin interogare:

```
delete data
{
graph :FacebookRelations {:George :hasFriend :Roxanne}
}
```

Verificați graful *FacebookRelations* pentru a observa faptul că o anumită afirmație a fost ștearsă.

Atenție la ștergerile care nu specifică un anumit graf! Următorul exemplu adaugă aceeași afirmație în trei grafuri: două identificate plus cel implicit:

```
insert data
{
graph :g1 {:Vienna :capitalOf :Austria}
graph :g2 {:Vienna :capitalOf :Austria}
:Vienna :capitalOf :Austria
}
```

Ați putea fi tentați să credeți că următoarea comandă va șterge doar afirmația din graful implicit:

```
delete data
{:Vienna :capitalOf :Austria}
```

În schimb, va șterge afirmațiile din toate grafurile (nu va mai exista nici o afirmație despre Viena)!
Important: dacă la INSERT nu se indică graful, inserarea are loc în graful implicit (fără identificator). Dacă la DELETE nu se indică graful, ștergerea este făcută în toate grafurile.

În graful *FacebookRelations*, să se șteargă toate afirmațiile ce îi au pe prietenii lui Anna drept subiect (ar trebui să dispară o singură afirmaţie avându-l pe Andrew ca subiect):

```
with :FacebookRelations
delete {?x ?y ?z}
where
{:Anna :hasFriend ?x.?x ?y ?z}
```

Clauza WITH oferă o prescurtare pentru a indica același graf atât în clauza de ștergere cât și în clauza WHERE. Cu alte cuvinte, aceeaşi interogare s-ar putea scrie şi astfel:

```
delete
{graph :FacebookRelations {?x ?y ?z}}
where
{graph :FacebookRelations {:Anna :hasFriend ?x.?x ?y ?z}}
```

Este important de observat că **şablonul de ştergere se aplică pe rezultatele şablonului WHERE**: mai întâi se selectează setul de cunoştinţe {:Anna :hasFriend ?x.?x ?y ?z} şi doar pe acest set are loc ştergerea!

Din acest motiv următoarea ștergere nu are niciun efect:

```
with :FacebookRelations
delete {?x ?y ?z}
where {:Anna :hasFriend ?x}
```

Motivul este că setul de cunoştinţe obţinute cu {:Anna :hasFriend ?x} nu va conţine nicio afirmaţie care să aibă pe ACELAŞI ?x ca subiect, deci {?x ?y ?z} nu va găsi nimic de şters în rezultatele şablonului WHERE.

Modificarea afirmaţiilor cu interogări SPARQL

Să se înlocuiască proprietatea *hasFriend* cu proprietatea *knows* în graful *FacebookRelations*:

```
with :FacebookRelations
delete
{?x :hasFriend ?y}
insert
{?x :knows ?y}
where
{?x :hasFriend ?y}
```

Observații:

- Nu este disponibilă o comandă UPDATE! Va trebui să folosiți astfel de combinaţii între comenzile DELETE și INSERT!
- Aceste combinaţii formează o singură operaţie şi nu două interogări succesive! De aceea nu trebuie să vă lăsaţi induşi în eroare de faptul că DELETE apare înainte de INSERT (execuţia propriu-zisă are loc în ordinea inversă apariţiei şabloanelor: mai întâi selecţia cu WHERE, apoi inserarea relaţiei noi, apoi ştergerea relaţiei vechi).

Distrugerea grafurilor cu SPARQL

Să se elimine graful *FacebookRelations*:

```
drop graph :FacebookRelations
```

Să se elimine graful implicit:

```
drop default
```

Prin eliminarea ultimelor grafuri, baza de cunoştinţe ar trebui să rămână goală.
Alte opțiuni disponibile sunt: DROP ALL (va distruge toate datele, nu o executați decât pentru a goli baze de cunoştinţe!)

4.4 Interogări la distanță

Interogările la distanţă sunt cele prin care se interoghează prin HTTP un alt server decât cel instalat local. De regulă e vorba de un serviciu public de interogare (serviciu de tip "SPARQL endpoint"). Avem şi posibilitatea de a interoga mai multe servere simultan, caz în care vorbim de **interogări federative**, deci despre **baze de cunoştinţe federative**.

Un exemplu de serviciu public de interogare SPARQL este cel oferit de baza de cunoştinţe DBPedia (versiunea RDF a lui Wikipedia):
http://dbpedia.org/sparql

Veți folosi formularul HTML disponibil la această adresă pentru a extrage o listă cu 50 de companii:

```
select * where
{
?x a <http://dbpedia.org/ontology/Company>
}
limit 50
```

Aplicăm limita de 50 pentru a evita blocarea sau așteptarea îndelungată (unele baze de cunoștințe publice sunt disponibile doar pentru testare, iar interogările cu multe rezultate pot fi respinse sau pot avea timpi foarte mari de execuţie; pentru a evita astfel de riscuri vom limita în unele cazuri numărul de rezultate aşteptat). Un click pe oricare din rezultate de afişează fragmentul de graf ce conţine afirmaţiile în care acel rezultat este subiect. Analizând aceste rezultate putem intui forma pe care o au termenii din DBPedia:

- Exemplu de clasă: http://dbpedia.org/ontology/Company;
- Exemple de proprietăţi: http://dbpedia.org/property/sponsor, http://dbpedia.org/ontology/closeTo; cele prefixate cu *ontology* sunt considerate mai generale (aplicabile la mai multe clase), cele prefixate cu *property* sunt specifice unui anumit domeniu sau unei singure clase;
- Exemplu de instanţă: http://dbpedia.org/resource/Hasbro

Pentru un studiu mai detaliat, setul de date complet poate fi descărcat de la următoarea adresă:
http://wiki.dbpedia.org/Downloads2015-10
(este posibil ca adresa URL să se schimbe de fiecare dată când apare o nouă versiune, de obicei de 1-2 ori pe an – pentru cea mai recentă versiune consultaţi meniul *Services and Resources – Datasets* de pe site-ul Dbpedia.org)

Explicaţii cu privire la structura setului de date sunt oferite la:
http://wiki.dbpedia.org/services-resources/datasets/dbpedia-datasets

Explicaţii cu privire la terminologie (vocabularul de clase şi proprietăţi folosite în DBPedia) sunt oferite la (sunt incluse acolo şi linkuri pentru a descărca doar terminologia separat de restul afirmaţiilor):
http://wiki.dbpedia.org/services-resources/ontology

Se oferă şi posibilitatea de a naviga prin terminologie, pentru o mai bună înţelegere a proprietăţilor ce pot fi interogate pentru fiecare tip de resursă:
http://mappings.dbpedia.org/server/ontology/classes/

Pentru interogarea de mai sus, am presupus cunoscut faptul că identificatorul pentru conceptul (clasa) Company este <http://dbpedia.org/ontology/Company>. Dacă nu îl ştim, putem solicita o listă cu toate clasele disponibile:

```
select ?c where
{
?x a ?c
}
```

Undeva în lista de rezultate putem găsi identificatorul pentru conceptul Company. Efectuați clic pe acesta pentru a vedea proprietățile afișate într-o pagină formatată cu HTML.

dbpedia.org/ontology/Company

About: compañía

An Entity of Type : Class, from Named Graph : http://dbpedia.org/resource/classes#, within Data Space : dbpedia.org

Property	Value
rdf:type	owl:Class
rdfs:isDefinedBy	http://dbpedia.org/ontology/
rdfs:label	compañía
rdfs:subClassOf	dbo:Organisation
wdrs:describedby	dbo:data/definitions.ttl
http://www.w3.org/ns/prov#wasDerivedFrom	http://mappings.dbpedia.org/index.php/OntologyClass:Company
is http://open.vocab.org/terms/defines of	http://dbpedia.org/ontology/
is http://open.vocab.org/terms/describes of	dbo:data/definitions.ttl
is rdfs:domain of	dbo:assetUnderManagement dbo:assets dbo:codeStockExchange dbo:equity dbo:fate dbo:groundsForLiquidation dbo:industry dbo:internationally dbo:marketCapitalisation dbo:netIncome dbo:operatingIncome dbo:production dbo:registration dbo:service dbo:stockExchange dbo:subsidiary
is rdfs:range of	dbo:architecturalBureau dbo:chain dbo:contractor dbo:designCompany dbo:distributingCompany dbo:firstPublisher dbo:owningCompany dbo:parentCompany dbo:producedBy dbo:productionCompany dbo:publisher dbo:subsidiary
is rdfs:subClassOf of	dbo:Publisher dbo:Airline dbo:Bank dbo:Brewery dbo:BusCompany dbo:Caterer dbo:LawFirm dbo:RecordLabel dbo:Winery

Figura 32 Conceptul Company în DBPedia

Această pagină permite utilizatorului să se deplaseze în cadrul grafului DBPedia prin clickuri pe identificatorii URI (posibilitatea de a da click pe un URI pentru a obţine o pagină HTML e asigurată de mecanismul de **dereferenţiere URI**). Prin această navigare putem observa şi alţi identificatori pentru diverse proprietăţi şi lucruri conectate la conceptul Company.

Realizați aceeași interogare în interfaţa RDF4J, dar în aşa fel încât RDF4J să forwardeze interogarea spre DBPedia. Sintaxa este similară cu cea în care foloseam clauza GRAPH, cu deosebirea că în loc de a indica graful local ca sursă de date se va indica serviciul de interogare DBPedia:

```
select * where
{
service <http://dbpedia.org/sparql>
{?x a <http://dbpedia.org/ontology/Company>}
}
limit 50
```

A se observa că în această interogare nu folosim prefixe, deoarece am folosit URI compleţi.

Să se obţină toate afirmaţiile DBPedia care au conceptul Company ca subiect sau obiect:

```
select * where
{
service <http://dbpedia.org/sparql>
{
{<http://dbpedia.org/ontology/Company> ?p ?o}
union
```

```
{?s ?p <http://dbpedia.org/ontology/Company>}
}
}
```

Să se extragă toate filmele regizate de James Cameron conform DBPedia (de aici încolo vom include prefixe manual; ele nu pot fi detectate automat de RDF4J, deoarece nu am făcut un upload de cunoştinţe ci folosim interfaţa RDF4J doar pentru a forwarda interogări spre DBPedia):

```
prefix db: <http://dbpedia.org/property/>
select ?t where
{
service <http://dbpedia.org/sparql>
{?dir db:name "James Cameron"@en.
?mov db:director ?dir.
?mov db:name ?t}
}
```

Am presupus că știm următoarele:

- Numele de persoane și titlurile lucrurilor sunt atașate cu proprietatea <http://dbpedia.org/property/name>
- Regizorii de filme sunt atașați cu proprietatea <http://dbpedia.org/property/director>

Dacă nu le ştim, am putea afla aceste lucruri interogând în formularul DBPedia (dbpedia.org/sparql) informațiile legate de stringul ”James Cameron”:

```
select * where
{
?x ?y "James Cameron"@en
}
```

În rezultatele găsite putem găsi identificatorii pentru James Cameron și proprietățile acestuia.

Interogările federative

Pentru a avea interogări federative trebuie să extragem informații din cel puțin două servere diferite. Pentru aceasta, pe lângă DBPedia vom mai utiliza şi o bază de cunoştinţe proprie creată în RDF4J, cu informaţii parţial conectate la DBPedia.

Încărcați următorul exemplu (din fișierul *awards.trig*) în baza de cunoștințe denumită MyRepo:

```
@prefix : <http://expl.at#>.
:awards
{
:JamesCameron a :OscarWinner, :Director; :hasName "James Cameron"@en.
:TimBurton a :OscarNominee, :Director; :hasName "Tim Burton"@en.
:CristiPuiu a :CannesWinner, :Director; :hasName "Cristi Puiu"@en.
:StevenSpielberg a :OscarWinner, :Director; :hasName "Spielberg"@en
}
```

Să se extragă din DBPedia titlurile filmelor regizorilor despre care în graful nostru local (*awards*) se știe că au câștigat premii Oscar:

```
prefix : <http://expl.at#>
prefix db:<http://dbpedia.org/property/>
select ?dir ?t where
{
service <http://localhost:8080/rdf4j-server/repositories/MyRepo>
     {graph :awards {?x a :OscarWinner; :hasName ?n}}
service <http://dbpedia.org/sparql>
    {?dir db:name ?n.
    ?mov db:director ?dir.
    ?mov db:name ?t}
}
```

A se observa că:

- am atribuit prefixul db pentru proprietăți DBPedia. Prefixul poate fi definit manual în RDF4J pentru a evita tastarea sa completă pe viitor (*Namespaces*);
- serverul nostru local are o adresă predefinită (*<http://localhost:8080/rdf4j-server/repositories/MyRepo>*); aceasta înseamnă că RDF4J nu e doar un server local de cunoştinţe, ci oferă şi acea funcţionalitate de "serviciu public de interogare" pe care o oferă şi DBPedia şi care se poate accesa cu clauza SERVICE (cu menţiunea că adresa serviciului conţine rdf4j-server în loc de rdf4j-workbench);
- variabila comună conectează afirmaţii din graful local cu informațiile de pe DBPedia.

Rezultatele ar trebui să ofere titlurile lui James Cameron chiar dacă îl avem și pe Spielberg declarat câștigător Oscar. Asta deoarece doar în cazul lui James Cameron numele din graful nostru este identic cu numele stocat în DBPedia. În cazul lui Spielberg graful nostru are doar numele de familie, deci variabila comună ?n nu va realiza potrivirea de stringuri.

Dacă știm că numele folosite de noi pot fi incomplete putem aplica un filtru pentru a verifica dacă numele găsit pe DBPedia conține numele folosit în graful nostru.

```
prefix : <http://expl.at#>
prefix db:<http://dbpedia.org/property/>
select ?dir ?t where
{
service <http://localhost:8080/rdf4j-server/repositories/MyRepo>
     {graph :awards {?x a :OscarWinner; :hasName ?n1}}
service <http://dbpedia.org/sparql>
    {?dir db:name ?n2.
    ?mov db:director ?dir.
    ?mov db:name ?t.
    filter (contains(str(?n2),str(?n1)))}
}
```

Observați că acum folosim două variabile diferite (?n1, ?n2) pentru a extrage numele regizorilor, apoi aplicăm un filtru cu testul contains() pentru a le compara. Acum vor fi afișate și filmele lui Spielberg. Mai există și alte funcții pentru procesarea stringurilor care pot fi utilizate pentru a căuta diverse similarităţi ce pot apare între stringuri. Există şi instrumente automate de detectare a similarităţii[45]

Totuşi, similaritatea între stringuri nu va garanta că avem informații despre același lucru (poate sunt și alte persoane cu numele de familie"Spielberg", poate unele stringuri sunt scrise în limbi diferite şi nu se poate detecta automat similaritatea lor).

În continuare presupunem că stringurile cu nume nu sunt disponibile, sau nu ne putem baza pe ele pentru a asigura potrivirea de nume. În general, numele nu sunt adecvate pentru a asigura identitatea, de aceea e preferabil să folosim identificatori URI în acest scop.

Din acest motiv, în următorul exemplu nu ne bazăm pe potrivirea de nume, ci pe reutilizarea de identificatori oferiţi de DBPedia. Problema este că în graful nostru am creat deja o serie de URI pentru regizori şi nu dorim să îi înlocuim (de exemplu, ar putea exista aplicaţii software care depind deja de acei URI). În acest caz, legătura trebuie să fie realizată prin crearea unei **ontologii de aliniere** ("alignment ontology") care de obicei ia forma unui dicţionar de echivalenţe între URI de provenienţă diferită (pe lângă echivalenţe poate include şi alte relaţii semantice care să asigure "traversarea" interogărilor dintr-o bază de cunoştinţe spre cealaltă).

Următorul exemplu este un graf (*mappings*) ce conţine o astfel de ontologie de aliniere, asigurând echivalarea între URI din graful nostru şi cei din DBPedia. E nevoie să încărcăm acest fişier TriG (alignment.trig) în RDF4J, în aceeaşi bază de cunoştinţe:

[45] Un instrument pentru definirea regulilor de similaritate (similarity rules) este Silk disponibil la adresa http://wifo5-03.informatik.uni-mannheim.de/bizer/silk/

```
@prefix : <http://expl.at#>.
@prefix dbr: <http://dbpedia.org/resource/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:mappings
{
:JamesCameron owl:sameAs dbr:James_Cameron.
:TimBurton owl:sameAs dbr:Tim_Burton.
:CristiPuiu owl:sameAs dbr:Cristi_Puiu.
:StevenSpielberg owl:sameAs dbr:Steven_Spielberg
}
```

Să se extragă lista cu regizorii din graful nostru, alături de identificatorii lor din DBPedia precum și lista cu titlurile acestora preluată tot din DBPedia:

```
prefix : <http://expl.at#>
prefix db:<http://dbpedia.org/property/>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?localdir ?dbpdir ?t where
{
service <http://localhost:8080/rdf4j-server/repositories/MyRepo>
      {
      graph :awards {?localdir a :Director}
      graph :mappings {?localdir owl:sameAs ?dbpdir}
      }
service <http://dbpedia.org/sparql>
     {?mov db:director ?dbpdir.
     ?mov db:name ?t}
}
```

Observați cum graful *mappings* asigură "puntea" între graful local *awards* şi cunoştinţele accesate la distanță din DBPedia.

Remarcați că fiecare server poate să găzduiască mai multe grafuri, iar interogările pot să urmărească afirmaţii ce conectează grafuri repartizate pe mai multe servere! Grafurile RDF distribuite pe mai multe servere şi conectate într-o astfel de manieră, care să permită "traversarea" interogărilor de la un server la altul, poartă numele **Linked Data** (date conectate), făcând astfel o distincţie importantă faţă de datele izolate! Dacă aceste grafuri sunt publice (oferă acces nerestricţionat la serviciul de interogare), avem de a face cu **Linked Open Data**. Dacă ele sunt accesibile doar în interiorul unei organizaţii şi partenerii direcţi ai acesteia, vorbim de **Linked Enterprise Data**.

Există situaţii când dorim să verificăm dacă există informații contradictorii în surse diferite (sau organizații). Pentru exemplificare, încărcați următoarele afirmaţii (disponibile în fișierul birth.trig) în RDF4J:

```
@prefix : <http://expl.at#>.
@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
:birthdata
{
dbr:Jim_Jarmusch dbo:birthPlace :Austria.
dbr:James_Cameron dbo:birthPlace dbr:Canada.
dbr:Ohio :isBirthPlaceOf dbr:Steven_Spielberg
}
```

Remarcați că folosim URI cunoscuți din DBPedia (prefixaţi cu dbr sau dbo), cu două excepţii:

- pentru Austria (declarată ca loc de naștere pentru Jim Jarmusch)
- pentru proprietatea cu care declarăm locul de naștere a lui Spielberg (isBirthPlaceOf).

Extrageți lista cu regizorii care au locul de naștere declarat atât în graful nostru local cât și în DBPedia. Pentru aceia la care se găsește două locuri de naștere diferite să se genereze un mesaj:

```
prefix : <http://expl.at#>
prefix dbo:<http://dbpedia.org/ontology/>
```

```
select ?dir ?bp1 ?bp2 (if(?bp1=?bp2,"valid","contradiction") as ?check)
where
{
service <http://localhost:8080/rdf4j-server/repositories/MyRepo>
     {graph :birthdata {?dir dbo:birthPlace ?bp1}}
service <http://dbpedia.org/sparql>
     {?dir dbo:birthPlace ?bp2}
}
```

Se va afișa contradicție pentru Jarmusch. James Cameron va avea mai multe rezultate deoarece DBPedia folosește aceeași proprietate pentru a atribui țara de naștere, statul sau regiunea de naștere. Printre acestea, se poate vedea rezultatul valid (dbr:Canada). Spielberg nu se regăsește printre rezultate deoarece am folosit o altă proprietate pentru a declara locul său de naștere.

Pentru a include și verificarea valorilor pentru Spielberg trebuie să adăugăm în ontologia de aliniere o echivalare între relaţia dbo:birthPlace și :isBirthPlaceOf. Atenţie însă, nu e vorba de o echivalare totală, deoarece la :isBirthPlaceOf persoana apare ca subiect şi locul ca obiect, în timp ce la dbo:birthPlace poziţiile sunt inversate. Din acest motiv echivalarea nu se face cu owl:sameAs, ci cu o altă relaţie standardizată, tot din vocabularul OWL. Încărcaţi în RDF4J fişierul cu conţinutul următor (din fișierul inverse.trig):

```
@prefix : <http://expl.at#>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:mappings
          {:isBirthPlaceOf owl:inverseOf dbo:birthPlace}
```

Reamintim că ontologiile de aliniere pot conţine şi alte "punţi" decât simpla echivalare de indivizi. În acest exemplu avem o mapare între o proprietate şi inversul acesteia, folosind relaţia standard **owl:inverseOf**. Acum putem să includem și verificarea pentru locul de naștere a lui Spielberg, incluzând noua mapare în interogare:

```
prefix : <http://expl.at#>
prefix dbo:<http://dbpedia.org/ontology/>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?dir ?localpl ?dbppl where
{
service <http://localhost:8080/rdf4j-server/repositories/MyRepo>
     {
     graph :mappings {?invprop owl:inverseOf dbo:birthPlace}
     graph :birthdata {
              {?dir dbo:birthPlace ?localpl}
              union
              {?localpl ?invprop ?dir}
              }
     }
service <http://dbpedia.org/sparql>
     {?dir dbo:birthPlace ?dbppl}
}
```

(acum comparația între locul de naștere găsit în DBPedia (?dbppl) și cel local (?localpl) este doar vizuală; nu am mai generat un mesaj de comparație pentru a păstra exemplul simplu)

Exemplele oferite aici sunt nu singurele modalități prin care putem realiza interogări federatie. În loc să avem mai multe opțiuni SERVICE/GRAPH în aceeași interogare, putem executa mai multe interogări separat și să transmitem între ele valorile relevante în mod manual sau prin programare (variabile intermediare în limbajul de programare din care executăm interogările). În continuare vom descompune interogarea bazată pe echivalări owl:sameAs.

Într-o primă interogare, preluăm din ontologia de aliniere identificatorii DBPedia pentru regizorii din graful nostru local:

```
prefix : <http://expl.at#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select?dbpdir where
{
service <http://localhost:8080/rdf4j-server/repositories/MyRepo>
      {
      graph :awards {?localdir a :Director}
      graph :mappings {?localdir owl:sameAs ?dbpdir}
      }
}
```

În a doua fază construim interogarea trimisă către DBPedia (pentru obţinerea filmelor) dar de data aceasta lista de regizori o includem manual:

```
prefix db: <http://dbpedia.org/property/>
prefix dbr: <http://dbpedia.org/resource/>
select ?dbdir ?t where
{
service <http://dbpedia.org/sparql>
             {?mov db:director ?dbdir.
             ?mov db:name ?t}
}
bindings ?dbdir
{
             (dbr:James_Cameron)
             (dbr:Tim_Burton)
             (dbr:Cristi_Puiu)
             (dbr:Steven_Spielberg)
}
```

Pentru fiecare identificator din lista BINDINGS, se va reexecuta interogarea spre DBPedia. Practic avem un ciclu FOR inclus manual în interogare. Astfel ni se permite să descompunem o interogare federativă, într-o succesiune de interogări mai simple. În cazuri realiste, lista BINDINGS poate fi destul de lungă și va fi creată prin programare (concatenări într-un ciclu FOR), în limbajul în care se creează aplicaţia client ce iniţiază astfel de interogări.

5. Procesarea grafurilor RDF în PHP

Biblioteca **EasyRDF**[46] se instalează în PHP cu ajutorul programului Composer (recitiţi explicaţiile oferite despre Composer, în capitolul despre cereri asincrone prin GuzzleHttp - acolo s-a explicat modul de instalare pentru Composer, în caz că nu îl aveţi deja instalat).

Presupunând că aveţi instalat Composer, porniţi linia de comandă Windows şi navigaţi la folderul site1, unde vom crea exemple cu EasyRDF:

```
cd c:\xampp\htdocs\site1
```

În folderul respectiv instalaţi EasyRDF cu ajutorul Composer, rulând comanda:

```
composer require easyrdf/easyrdf
```

(comanda va începe descărcarea fişierelor EasyRDF de pe Internet şi stocarea lor într-un folder numit vendor, de unde vor trebui importate în scripturile PHP)

Documentaţia completă a claselor EasyRDF e disponibilă la:
http://www.easyrdf.org/docs/api

Procesare de grafuri din surse locale

În primul exerciţiu vom încărca un graf RDF dintr-un fişier Turtle extern, stocat tot în Apache (site1). Salvaţi următorul fişier în site1, cu numele exemplu.ttl (verificaţi să fie accesibil în browser la adresa http://site1.com/exemplu.ttl):

```
@prefix : <http://expl.ro#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

:Ana a foaf:Person, :Student; foaf:knows :Alin, :Andrei; foaf:name "Pop Ana"; :studiazaLa :UBB; foaf:age 20.
:Alin a foaf:Person, :Programator; foaf:knows :Maria; foaf:name "Ionescu Alin"; :lucreazaLa :Endava.
:Andrei a foaf:Person; foaf:name "Popescu Andrei".
:Maria a foaf:Person; foaf:name "Mihaila Maria".
:Endava a foaf:Organization; rdfs:label "Endava SRL".
:UBB a foaf:Organization; rdfs:label "Universitatea Babes-Bolyai".
```

Observaţii:

- Am utilizat o serie de concepte din vocabularul FOAF, folosit frecvent în descrierea de persoane şi relaţii sociale:
  - foaf:Person e clasa "persoană" (mulţimea tuturor persoanelor);
  - foaf:Organization e clasa "organizaţie" (mulţimea tuturor organizaţiilor);
  - foaf:name e proprietate prin care alocăm un nume de tip string (de obicei persoanelor, dar nu numai);
  - foaf:knows e relaţia prin care două persoane se cunosc între ele;
  - foaf:age e proprietatea "vârstă", cu valori integer;
  - alte concepte FOAF pot fi consultate la http://xmlns.com/foaf/spec/
- Am utilizat şi concepte RDF standard:
  - Litera "a" (verbul "a fi") prin care spunem ce sunt diverse entităţi menţionate: Ana e o persoană şi un student, Alin e o persoană şi un programator, mai avem şi alte persoane, avem şi două organizaţii;
  - rdfs:label e similar cu foaf:name, se foloseşte pentru a asocia o etichetă de tip string unei entităţi; e mai generală decât foaf:name, deoarece eticheta respectivă nu trebuie neapărat să fie un nume (poate fi poreclă, titlu sau pur şi simplu o versiune afişabilă mai uşor de citit decât URI-ul entităţii);
- Restul conceptelor sunt improvizate de noi, cu prefix vid – clasele Student, Programator, proprietăţile lucreazaLa, studiazaLa.

---

[46] http://www.easyrdf.org/

Mai departe, creaţi în acelaşi folder site1, scriptul easyrdf1.php (explicaţiile sunt incluse în cod):

```
<?php
require 'vendor/autoload.php';
// am importat clasele EasyRDF din folderul vendor în care au fost instalate (autoload le importă pe măsură ce ele devin necesare);
$graf=new EasyRdf_Graph();
// am instanţiat un graf gol
$graf->load("http://site1.com/exemplu.ttl","turtle");
// am încărcat graful cu informaţia din fişierul Turtle
$prefixe=new EasyRDF_Namespace();
// am instanţiat colecţia de prefixe ale grafului (aceasta va crea o serie de prefixe populare sau standardizate: foaf, rdf, rdfs etc.)
$prefixe->set("","http://expl.ro#");
// am inclus prefixul propriu în colecţie

print "<b>Prefixele inregistrate sunt:</b>";
print_r($prefixe->namespaces());
print "<br/><br/>";
// am printat toate prefixele disponibile – observaţi o lungă listă de prefixe predefinite, plus prefixul gol introdus de noi
// printarea s-a făcut cu print_r, deoarece lista e un array, nu un string (am putea şi să aplicăm json_encode pentru a-l vedea ca string)


print "<b>Continutul grafului in format N-triples este urmatorul (verificati in View page source!):</b>";
print $graf->serialise("ntriples");
print "<br/><br/>";
// cu serialise() am convertit întreg graful în sintaxa Ntriples
// datorită tag-urilor, nu puteţi vedea conţinutul direct în browser, ci doar în sursa paginii)
// consultaţi şi alte formate de serializare în documentaţia oficială
// atenţie, pentru conversie în JSON-LD e necesară instalarea, tot prin Composer, a extensiei ml/json-ld

print "<b>Continutul grafului in format HTML este urmatorul:</b>";
print $graf->dump();
// metoda dump() generează o pagină HTML ce prezintă conţinutul grafului în browser într-o manieră uşor de citit de către om
?>
```

În următoarele exemple vom extrage diverse informaţii din graf cu ajutorul metodelor obiectuale oferite de EasyRDF:

```
<?php
require 'vendor/autoload.php';
$graf=new EasyRdf_Graph();
$graf->load("http://site1.com/exemplu.ttl","turtle");
$prefixe=new EasyRDF_Namespace();

$persoane=$graf->allOfType("foaf:Person");
print "<b>Persoanele identificate sunt:</b><br/>";
foreach ($persoane as $persoana)
        {
        print $persoana."<br/>";
        }
?>
```

Acest exemplu foloseşte funcţia allOfType() pentru a obţine toate instanţele de tip persoană. Clasa persoană e indicată în forma prescurtată foaf:Person, deoarece prefixul foaf e recunoscut automat de EasyRDF (alături de celelalte listate în exemplul precedent).
Funcţia allOfType() returnează un vector de URI ce trebuie parcurs cu un ciclu foreach pentru a afişa lista persoanelor.

Adăugaţi la acelaşi fişier (ca să nu mai repetăm iniţializarea) următoarele:

```
$prefixe->setDefault("http://expl.ro#");
$studenti=$graf->allOfType("Student");
print "<b>Studentii identificati sunt:</b><br/>";
foreach ($studenti as $student)
        {
        print $student."<br/>";
        }
```

Funcţia obţine toate instanţele de tip Student. Observaţi că, declarând http://expl.ro# ca prefix implicit (cu setDefault), suntem scutiţi de a mai scrie prefixul clasei Student. Dacă nu includem acea declaraţie, suntem nevoiţi să scriem $graf->allOfType("http://expl.ro#Student"), pentru că expl.ro nu are un prefix recunoscut automat de EasyRDF.

Adăugaţi mai departe:

```
$resurse=$graf->resources();
print "<b>Lista completa a resurselor identificate, indiferent de tip:</b><br/>";
foreach ($resurse as $resursa)
        {
        print $resursa."<br/>";
        }
```

Funcţia resource() obţine toate resursele prezente în graf.

Adăugaţi mai departe:

```
$tipuriAna=$graf->typesAsResources("Ana");
print "<b>Lista claselor in care e inclusa Ana:</b><br/>";
foreach ($tipuriAna as $tip)
        {
        print $tip."<br/>";
        }
```

Funcţia typesAsResources() obţine toate tipurile entităţii http://expl.ro#Ana (putem scrie direct Ana datorită declarării de prefix implicit de mai înainte[47]).

Adăugaţi mai departe:

```
print "<br/><b>Este Ana student?</b><br/>";
print $graf->isA("Ana","Student");
print "<br/><b>Ce nume are Endava?</b><br/>";
print $graf->label("Endava");
print "<br/><b>Ce nume are Ana?</b><br/>";
print $graf->label("Ana");
```

Funcţia isA() testează dacă o entitate are un anumit tip şi returnează o valoare booleană
Funcţia label() returnează o etichetă afişabilă care s-a asociat unei entităţi – această e culeasă fie din foaf:name (cazul Ana), fie din rdfs:label (cazul Endava). Dacă ambele sunt prezente rdfs:label are prioritate.

Adăugaţi mai departe:

```
print "<br/><b>Ce proprietati are Ana?</b><br/>";
$proprietati=$graf->propertyUris("Ana");
foreach ($proprietati as $proprietate)
        {
        print $proprietate."<br/>";
        }
```

Funcţia propertyUris() returnează toate proprietăţile unui anumit subiect, aici Ana.

Adăugaţi mai departe:

```
print "<br/><b>Pe cine cunoaste Ana si unde lucreaza cunoscutii sai?</b><br/>";
$cunoscuti=$graf->allResources("Ana","foaf:knows");
foreach ($cunoscuti as $cunoscut)
        {
```

[47] toate exemplele ce urmează se bazează pe prefixul implicit aşa că dacă le copiaţi într-un fişier separat, asiguraţi-vă că includeţi şi declaraţia respectivă (alături de iniţializarea grafului)

```
print $cunoscut;
if ($cunoscut->hasProperty("lucreazaLa"))
        {print " are locul de munca: ".$cunoscut->lucreazaLa;}
else
        {print " nu are loc de munca declarat";}
print "<br/>";
}
```

Cu funcţia allResources() am obţinut toate entităţile cunoscute de Ana. Apoi am parcurs lista aceasta şi am testat cu hasProperty() dacă acei cunoscuţi au declarat locul de muncă:

- Pentru cei care au, locul de muncă e afişat accesând direct, prin sintaxa obiectuală, proprietatea lucreazaLa. Atenţie acest mod comod de acces e posibil doar pentru proprietăţile cu prefixul declarat ca implicit (cu setDefault). Altfel, valorile proprietăţilor se accesează cu $cunoscut->get("lucreazaLa"), iar dacă suntem nevoiţi să folosim URI complet, cu $cunoscut->get("<http://expl.ro#lucreazaLa>") – atenţie la <>
- Pentru cei care nu au loc de muncă se generează doar un mesaj.

Adăugaţi mai departe:

```
print "<br/><b>Cum ii cheama pe cei cunoscuti de Ana?</b><br/>";
$toateNumele=$graf->all("Ana","foaf:knows/foaf:name");
foreach ($toateNumele as $nume)
        {
        print $nume."<br/>";
        }
```

Cu funcţia all() am obţinut toate valorile la care se ajunge mergând de la Ana pe lanţul foaf:knows/foaf:name. Cu alte cuvinte, am accesat toate numele cunoscuţilor Anei[48].

Adăugaţi mai departe:

```
print "<br/><b>Ce URI are entitatea cu numele Popescu Andrei?</b><br/>";
$subiecte=$graf->resourcesMatching("foaf:name","Popescu Andrei");
foreach ($subiecte as $subiect)
        {
        print $subiect."<br/>";
        }
```

De data aceasta am parcurs graful în sens opus relaţiei foaf:name: cunoscându-se valoarea numelui, am returnat URI-ul subiectului care are acel nume, cu resourcesMatching().

O altă metodă de a accesa informaţii mergând în sens opus proprietăţilor din graf este exemplificat mai jos.

Adăugaţi mai departe:

```
print "<br/><b>Cine lucreaza la Endava?</b><br/>";
$Endava=$graf->resource("Endava");
$angajati=$Endava->allResources("^lucreazaLa");
foreach ($angajati as $angajat)
        {
        print $angajat."<br/>";
        }
```

Exemplul returnează angajaţii de la Endava, mergând dinspre obiectul Endava spre subiectele relaţiei lucreazaLa.

- Mai întâi am selectat în mod izolat entitatea Endava cu funcţia resource();

[48] în general în browser trebuie să afişăm nume şi etichete, deci nu URI cum am afişat în majoritatea exemplelor!

- Apoi cu funcţia allResources() i-am cules toate entităţile conectate, mergând în sensul opus al relaţiei lucreazaLa; observaţi folosirea caracterului ^ pentru a indica sensul opus (dinspre obiect spre subiect).

Atenţie, atunci când accesăm valori literale de proprietăţi (string, integer etc.) şi nu URI, funcţiile EasyRDF pot să difere.
Adăugaţi mai departe:

```
print "<br/><b>Ce varsta are Ana?</b><br/>";
print $graf->getLiteral("Ana","foaf:age");
print "<br/><b>Unde studiaza Ana?</b><br/>";
print "URI complet: ".$graf->getResource("Ana","studiazaLa")."</br>";
print "URI relativ: ".$graf->getResource("Ana","studiazaLa")->localName()."</br>";
print "<br/><b>Care e prefixul lui foaf:name?</b><br/>";
print $graf->resource("foaf:name")->prefix();
```

Observaţi folosirea lui getLiteral() când dorim să extragem o valoare simplă, comparativ cu getResource care returnează un URI. Acel URI poate fi apoi descompus cu ajutorul localName() care ne dă varianta scurtă neprefixată (URI relativ), sau cu prefix() care ne dă prefixul[49].

De asemenea, mai observaţi şi că funcţiile care încep cu **get** (getLiteral,getResource) returnează o singură valoare, chiar dacă există mai multe (de exemplu rdfs:label poate asocia mai multe denumiri scrise în mai multe limbi). Pentru a obţine liste complete (care apoi trebuie parcurse în cicluri FOR), se folosesc funcţiile ce încep cu **all** (allResources,all,allLiterals).

Toate exemplele de până acum au arătat operaţii de citire dintr-un graf care exista stocat pe disc, într-un fişier. Exemplul următor include operaţii de scriere, prin care se adaugă la graful respectiv informaţii noi. Creaţi un fişier nou în acest scop:

```
<?php
require 'vendor/autoload.php';
$graf=new EasyRdf_Graph();
$graf->load("http://site1.com/exemplu.ttl","turtle");
$prefixe=new EasyRDF_Namespace();
$prefixe->setDefault("http://expl.ro#");
// până aici a fost partea de iniţializare, folosită întotdeauna
$graf->addResource("Irina","foaf:knows","Petru");
$graf->addResource("Irina","foaf:knows","Pavel");
$graf->add("Irina","foaf:age","22");
print $graf->dump();
$Irina=$graf->resource("Irina");
$Irina->areSalariu=10000;
print $graf->dump();
$graf->delete("Irina","foaf:age");
print $graf->dump();
$graf->deleteResource("Irina","foaf:knows","Pavel");
print $graf->dump();
print $graf->serialise("json");
?>
```

Observaţii:

- Cu addResource() am adăugat afirmaţii noi, având ca obiect un URI;
- Cu add() am adăugat o afirmaţie nouă, având ca obiect o valoare simplă;
    - După aceste operaţii cu dump() am afişat graful integral în care puteţi remarca noile informaţii despre Irina
- Apoi am selectat în mod izolat entitatea Irina cu resource() şi i-am adăugat o proprietate nouă (areSalariu) cu valoare simplă. După cum am comentat şi la get(), am folosit sintaxa obiectuală

[49] căutaţi în documentaţie şi clasa EasyRdf_ParsedUri care oferă şi metode mai avansate de descompunere a unui URI în părţi componente (partea de protocol, partea de server, partea de cale etc.)

Irina->areSalariu pentru a genera o proprietate cu prefixul declarat ca implicit. Pentru altele, de exemplu pentru schimbarea numelui, am folosit:

```
$Irina->set("foaf:name","Pop Irina")
```

sau, dacă Irina nu ar fi fost selectată în prealabil (cu resource(), ori returnată de alte funcţii), am folosi:

```
$graph->set("Irina","foaf:name","Pop Irina")
```

(cu această ocazie amintim că multe dintre funcţiile prezentate aici pot fi accesate atât prin intermediul obiectului graf, cât şi prin intermediul unui subiect anterior selectat).
  - Urmează din nou un dump() cu care putem remarca modificările
- Apoi cu deleteResource() am şters una din afirmaţiile despre Irina
  - Urmează din nou un dump() cu care putem remarca modificările
- În final am serializat graful în format JSON şi l-am printat (de exemplu pentru a fi trimis ca răspuns unei cereri AJAX/HTTP sosită de la client, răspuns ce va trebui procesat precum orice alt răspuns JSON discutat până acum)

Procesarea grafurilor obţinute de la servicii de cunoştinţe compatibile cu standardul SPARQL HTTP

Serviciile de cunoştinţe sunt servere RDF ce oferă o interfaţă REST de interogare la distanţă, prin HTTP. Această interfaţă poate oferi câteva adrese predefinite, la care serviciul răspunde în diverse moduri, sau poate accepta chiar interogări SPARQL trimise de la un client HTTP. Pentru a iniţia astfel de cereri şi interogări, EasyRDF oferă două clase:

- **EasyRdf_Sparql_Client** se poate conecta la orice serviciu care respectă protocolul SPARQL HTTP standard, oferind funcţii pentru executarea interogărilor SPARQL de orice fel;
- **EasyRdf_Http_Client** se poate conecta la orice server şi script, permiţând trimiterea de cereri HTTP configurate explicit în cele mai mici detalii (aşa cum am făcut şi cu cURL sau Guzzle).

Prima variantă e mai uşor de folosit, dar se bazează pe configurări predefinite (nu putem modifica antetul HTTP, metoda GET/POST/etc. ori parametri, putem doar trimite o interogare SPARQL bazându-ne pe faptul că serviciul va accepta configurările predefinite). *De regulă aceste configurări predefinite sunt suficiente, pentru că există tendinţa ca serverele RDF să fie compatibile cu standardul SPARQL HTTP* (ce impune configurările predefinite).

A doua variantă e mai dificil de folosit, dar se poate conecta şi la servere ce nu sunt compatibile cu standardul SPARQL HTTP, *permiţând o configurare amănunţită a cererilor HTTP*. În plus, cererile HTTP cu configurări explicite se pot folosi şi în comunicarea cu alte servere ori servicii ce nu au nicio legătură cu RDF (practic **EasyRdf_Http_Client** se comportă similar cererilor cURL, cu o sintaxă ceva mai simplă).
Majoritatea serviciilor de cunoştinţe publice oferă doar interfaţă de citire, dar pot avea şi interfaţă de scriere (ascunsă publicului). Exemplul de mai jos foloseşte serviciul SPARQL de la DBpedia:

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("https://dbpedia.org/sparql");
$interogare="describe dbr:JamesCameron";
$rezultat=$client->query($interogare);
print $rezultat->dump();
?>
```

Observaţi:

- Adresa accesibilă publicului este https://dbpedia.org/sparql. Are şi o versiune vizibilă în browser sub formă de formular HTML, în care se pot exersa interogări
- Folosim adresa serviciului în EasyRdf_Sparql_Client pentru a iniţializa un client pe care îl creăm în PHP;
- Interogarea SPARQL e memorată într-un string. Cea din exemplu e o interogarea DESCRIBE asupra unei resurse din DBpedia, în cazul nostru despre James Cameron; o serie de prefixe sunt recnoscute

automat, fără a mai fi nevoie să mai adăugăm declarațiile de spații de nume: cum ar fi prefix dbr:<https://dbpedia.org/resource>[50];

- Cu funcţia query am executat interogarea, iar cu dump am făcut rezultatul ei vizibil în browser (răspunsul interogărilor DESCRIBE e un obiect de tip EasyRdf_Graph, deci poate fi afişat cu dump, aşa cum am vizualizat grafuri şi în alte exemple precedente; atenţie însă, interogările SELECT nu răspund cu grafuri!)

Notă! este posibil să fie nevoie să instalați încă o bibliotecă pentru a permite procesarea datelor în format JSON-LD: În folderul site1, prin linia de comandă, o vom instala cu ajutorul programului Composer – composer require ml/json-ld

Următorul exemplu va extrage doar abstractul în limba engleză pentru resursa James Cameron:

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("https://dbpedia.org/sparql");
$interogare="select ?a where {dbr:James_Cameron dbo:abstract ?a filter (lang(?a)='en')}";
$rezultat=$client->query($interogare);
print $rezultat->dump();
?>
```

În continuare, pentru a putea exersa operaţii CRUD, avem nevoie de un serviciu de cunoştinţe care să permită şi operaţii de scriere. Un astfel de serviciu este inclus în RDF4J şi e accesibil la adresele următoare (testate cu RDF4J 2.2 rulând pe Tomcat 8.0):

- http://localhost:8080/rdf4j-server/repositories/numeleBD e adresa pentru interogări de citire
- http://localhost:8080/rdf4j-server/repositories/numeleBD/statements e adresa pentru interogări de scriere
- serviciul mai oferă şi alte adrese ce pot fi accesate prin cereri HTTP, însă doar cele de mai sus sunt preconfigurate să accepte cereri compatibile cu standardul SPARQL HTTP (aşa cum e necesar în EasyRdf_Sparql_Client()); celelalte adrese oferite de serviciu vor fi accesate prin cereri HTTP cu configurări explicite.

Porniţi serverul RDF4J şi creaţi pe acesta o bază de grafuri nouă, cu numele şi ID-ul "grafuri". Încărcaţi în ea următorul conţinut, scris în sintaxa TriG:

```
@prefix : <http://expl.ro#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

:grafPersoane {
   :Ana a foaf:Person, :Student; foaf:knows :Alin, :Andrei; foaf:name "Pop Ana"; :studiazaLa :UBB; foaf:age 20.
   :Alin a foaf:Person, :Programator; foaf:knows :Maria; foaf:name "Ionescu Alin"; :lucreazaLa :Endava.
   :Andrei a foaf:Person; foaf:name "Popescu Andrei".
   :Maria a foaf:Person; foaf:name "Mihaila Maria".}
:grafOrganizatii {
   :Endava a foaf:Organization; rdfs:label "Endava SRL".
   :UBB a foaf:Organization; rdfs:label "Universitatea Babes-Bolyai".}
```

În PHP interogaţi aceste grafuri, pentru a afişa toate informaţiile despre Alin. Folosim pentru asta DESCRIBE şi având în vedere că rezultatul e un obiect de tip graf, putem să aplicăm asupra lui toate metodele obiectuale ale clasei EasyRdf_Graph:

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("http://localhost:8080/rdf4j-server/repositories/grafuri");
$interogare="prefix : <http://expl.ro#> describe :Alin";
$rezultat=$client->query($interogare);
print $rezultat->dump();
//am afisat graful extras cu DESCRIBE
```

[50] pentru o listă complete cu prefixele recunoscute accesați http://dbpedia.org/sparql?help=nsdecl

```
$prefixe=new EasyRdf_Namespace();
$prefixe->setDefault("http://expl.ro#");
$cunoscuti=$rezultat->allResources("Ana","foaf:knows");
foreach ($cunoscuti as $cunoscut)
		{
		print $cunoscut."<br/>";
		}
//observati ca singurul cunoscut detectat e Alin (cautarea s-a facut pe rezultatul lui DESCRIBE, nu pe graful integral din RDF4J!)
?>
```

În următorul exemplu executăm o interogare CONSTRUCT ce generează relaţii noi (cunoastePe) în loc de foaf:knows şi construieşte un graf nou din noile relaţii (fără a-l salva pe server!). Prin urmare, la fel ca pentru DESCRIBE, şi aici rezultatul e un graf ce poate fi afişat sau manipulat mai departe cu ajutorul clasei EasyRdf_Graph:

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("http://localhost:8080/rdf4j-server/repositories/grafuri");
$interogare="prefix : <http://expl.ro#> construct {?x :cunoastePe ?y} where {{?x foaf:knows ?y}union{?y foaf:knows ?x}}";
$reteaCunoscuti=$client->query($interogare);
print $reteaCunoscuti->dump();
?>
```

În schimb interogările SELECT nu răspund cu grafuri bine formate, ci cu vectori de înregistrări ce trebuie parcurşi pentru a extrage datele relevante. În exemplul de mai jos realizăm două interogări SELECT:

- prima obţine toate subiectele şi numele ce le-au fost asociate prin foaf:name (observaţi că prefixul foaf nu l-am mai inclus în interogare pentru că îl recunoaşte automat serverul; alte servere sau alte prefixe ar putea să trebuiască incluse la începutul fiecărei interogări!);
- a doua interogare obţine lista claselor (tipurilor) prezente în graful :grafPersoane

În ambele cazuri un ciclu FOR parcurge rezultatele şi le concatenează în mesaje vizibile în browser.

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("http://localhost:8080/rdf4j-server/repositories/grafuri");
print "<b>Lista subiectelor si numelor:</b><br/>";
$interogare="select ?subiect ?nume where {?subiect foaf:name ?nume}";
$rezultate=$client->query($interogare);
foreach ($rezultate as $rezultat)
		{
		print "Subiectul ".$rezultat->subiect." are numele ".$rezultat->nume."<br/>";
		}

print "<b>Lista tipurilor din graful persoanelor:</b><br/>";
$interogare="select distinct ?tip where {graph <http://expl.ro#grafPersoane> {?subiect a ?tip}}";
$rezultate=$client->query($interogare);
foreach ($rezultate as $rezultat)
		{
		print $rezultat->tip."<br/>";
		}

print "<b>Lista grafurilor:</b><br/>";
$grafuri=$client->listNamedGraphs();
foreach ($grafuri as $graf)
		{print "Am gasit graful ".$graf."<br/>";}
?>
```

Observaţi că la final am realizat şi o listă a tuturor grafurilor detectate în baza "grafuri", obţinute prin listNamedGraphs().

În continuare realizăm interogări de inserare. Adresa la care serviciul RDF4J acceptă operaţii de scriere compatibile cu standardul SPARQL HTTP e puţin diferită (include termenul "statements"). Alte servere ar putea avea adrese identice pentru toate tipurile de operaţii, dar separarea lor este destul de uzuală, având în vedere că serviciile publice adesea nu doresc să ofere acces publicului pentru operaţii de scriere.

Am inserat câte un graf nou cu câteva afirmaţii, prin două metode:

- mai întâi o interogare INSERT transmisă prin funcţia update() spre adresa la care serviciul acceptă operaţii de scriere; în acest caz datele au fost prevăzute chiar în corpul interogării;
- în a doua variantă graful e iniţializat şi construit de la zero cu EasyRdf_Graph, apoi trimis fără nicio interogare, cu ajutorul funcţiei insert() oferită de clasa EasyRdf_Sparql_Client;
- în ambele variante am printat confirmări HTTP cu codul 204 (succes fără conţinut returnat).

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("http://localhost:8080/rdf4j-server/repositories/grafuri/statements");

//Aici inseram un graf nou cu o afirmatie noua prin intermediul interogarii INSERT
$interogare="prefix : <http://expl.ro#> insert data {graph :grafNou {:Andreea :lucreazaLa :ANAF}}";
print $client->update($interogare)."<br/>";

//Iar aici inseram un graf construit de la zero trimis ca pachet de date cu functia insert()
$graf=new EasyRdf_Graph("http://expl.ro#grafNou2");
$prefixe=new EasyRDF_Namespace();
$prefixe->setDefault("http://expl.ro#");
$graf->addResource("Irina","foaf:knows","Petru");
$graf->addResource("Irina","foaf:knows","Pavel");
$graf->add("Irina","foaf:age","22");
$client=new EasyRdf_Sparql_Client("http://localhost:8080/rdf4j-server/repositories/grafuri/statements");
print $client->insert($graf,"http://expl.ro#grafNou2");
?>
```

În urma execuţiei verificaţi în RDF4J apariţia celor două grafuri noi (la Contexts).
În continuare ştergem cele două grafuri noi, prin metode similare inserării lor:

- în prima variantă apelăm din nou la funcţia update(), care acum însă transmite o interogare DELETE WHERE;
- în a doua variantă apelăm la funcţia clear() oferită de clasa EasyRdf_Sparql_Client;
- a doua variantă e mai facilă, dar şterge grafuri întregi, în timp ce prima ne permite să trimitem interogări oricât de fine (de exemplu pentru a şterge o singură afirmaţie).

```
<?php
require 'vendor/autoload.php';
$client=new EasyRdf_Sparql_Client("http://localhost:8080/rdf4j-server/repositories/grafuri/statements");

// aici stergem primul graf inserat cu exemplul precedent, trimitand o interogare DELETE prin functia update()
$interogare="prefix : <http://expl.ro#> delete where {graph :grafNou {?x ?y ?z}}";
print $client->update($interogare)."<br/>";

// aici stergem al doilea graf inserat cu exemplul precedent, beneficiind de functia clear()
$rezultat=$client->clear("http://expl.ro#grafNou2");
print $rezultat;
?>
```

Procesarea grafurilor obţinute de la servicii de cunoştinţe ce nu sunt compatibile cu standardul SPARQL HTTP

Exemplificăm în continuare şi interacţiunea cu RDF4J în moduri care nu sunt compatibile cu standardul SPARQL HTTP dar permit totuşi interogări prin diverse combinaţii de antete HTTP, metode HTTP şi parametri codificaţi. Doar că în cazul cererilor necompatibile cu standardul SPARQL, nu mai putem beneficia de cererile preconfigurate oferite de clasa și funcțiile EasyRdf_Sparql_Client. Avem însă alte posibilităţi:

- să folosim **funcţiile generice de încărcare a unui fişier accesibil prin URL** (load() oferită de EasyRDF, file_get_contents() oferită nativ în PHP etc.); acestea nu vor permite configurări deoarece se bazează implicit pe metoda GET (eventual cu parametri alipiţi adresei serviciului);
- să folosim **EasyRdf_Http_Client**, caz în care putem realiza configurări detaliate explicit;

- să folosim orice alte **librării capabile să trimită cereri HTTP** (cURL, guzzle etc.), caz pe care nu îl vom mai detalia aici, deoarece au fost deja exemplificate cu alte ocazii.

Indiferent de caz, la construirea adreselor trebuie să urmărim documentaţia serviciului RDF4J, ce indică toate combinaţiile de adrese, metode HTTP şi parametri pentru a obţine diverse rezultate: http://docs.rdf4j.org/rest-api/

În varianta următoare folosim funcţia generică **load()** pentru a încărca într-un graf tot răspunsul primit de la adrese la care serviciul RDF4J răspunde cu grafuri (la cereri GET neconfigurate):

- în prima variantă am folosit adresa utilizată anterior la operaţii de scriere; dacă i se trimite o cerere GET neconfigurată, această adresă răspunde cu conţinutul integral al bazei de cunoştinţe!
- în a doua variantă am folosit o adresă mai complexă, compatibilă cu protocolul GraphStore[51], în care putem realiza filtrare după un graf, dacă includem URI-ul acestuia ca parametru GET; parametrul trebuie codificat prin urlencode deoarece în URI se găsesc caractere ce ar putea fi produce erori sau confuzie când apar în adrese URL.

```
<?php
require 'vendor/autoload.php';
$graf=new EasyRdf_Graph();

//observati ca RDF4J raspunde automat cu tot continutul bazei de cunostinte la adresa pentru operatii de scriere
print "<br/><b>Continutul integral:</b><br/>";
$graf->load("http://localhost:8080/rdf4j-server/repositories/grafuri/statements");
print $graf->dump();

$graf2=new EasyRdf_Graph();

//la adresa de mai jos, RDF4J permite filtrare dupa grafuri conform protocolului GraphStore
print "<br/><b>Continutul grafului organizatiilor:</b><br/>";
$graf2->load("http://localhost:8080/rdf4j-server/repositories/grafuri/rdf-graphs/service?graph=".urlencode("http://expl.ro#grafOrganizatii"));
print $graf2->dump();
?>
```

În varianta următoare realizăm o interogare SELECT. Aceasta necesită o configurare mai detaliată a cererii HTTP, în care trebuie să declarăm formatul MIME în care aşteptăm răspunsul.

```
<?php
require 'vendor/autoload.php';
print "<b>Din nou lista subiectelor si numelor, cu interogarea integrata intr-o cerere HTTP configurata:</b><br/>";
$adresa="http://localhost:8080/rdf4j-server/repositories/grafuri?query=";
$interogare=urlencode("prefix foaf: <http://xmlns.com/foaf/0.1/> select ?subiect ?nume where {?subiect foaf:name ?nume}");
$clienthttp=new EasyRdf_Http_Client($adresa.$interogare);
$clienthttp->setHeaders("Accept","application/sparql-results+json");

$rezultatJSON=$clienthttp->request()->getBody();
print "<br/><br/><b>Raspunsul JSON sosit este: </b>".$rezultatJSON;
$rezultatRestructurat=new EasyRdf_Sparql_Result($rezultatJSON,"application/sparql-results+json");
print "<br/><br/><b>Raspunsul restructurat este: </b>".$rezultatRestructurat;
?>
```

Declararea se face în antetul HTTP numit Accept. Cel mai comod format este SPARQL Results+JSON, al cărui cod MIME este application/sparql-results+json[52]. Analizaţi şi testaţi codul:

- observaţi modul de construire a adresei, adăugând interogarea în parametrul query, cu valoarea codificată prin urlencode() pentru a evita problemele cu anumite caractere;

[51] Protocolul GraphStore este o alternativă la standardul SPARQL HTTP care însă permite doar operaţii la nivel de grafuri întregi (deci fără interogări detaliate)

[52] Codurile MIME pentru diverse formate RDF se pot consulta la http://docs.rdf4j.org/rest-api/#_content_types

- observaţi utilizarea clasei EasyRdf_Http_Client; aceasta seamănă mai degrabă cu cURL decât cu EasyRdf_Sparql_Client, pentru că nu ne mai oferă funcţii directe de interogare, ci doar funcţii generice de configurare a cererii HTTP şi executare a sa:
    - un exemplu de astfel de funcţie este setHeaders() prin care am modificat antetul HTTP "Accept", indicând codul MIME al formatului în care aşteptăm răspunsul;
    - un alt exemplu e funcţia request() care execută cererea (prin metoda implicită GET) şi oferă răspunsul prin funcţia getBody();
- observaţi că am afişat răspunsul în două moduri:
    - prima dată am afişat structura JSON integrală care a sosit ca răspuns la SELECT;
    - apoi am folosit clasa EasyRdf_Sparql_Result pentru a converti răspunsul JSON într-o structură tabelară mai prietenoasă (verificaţi în View page source), similară celei în care am fi primit răspunsul dacă accesăm serviciul prin EasyRdf_Sparql_Client

Mai departe, datele ce prezintă interes pot fi scoase din oricare din cele două moduri de structurare, după cum se vede în următorul fragment de cod (copiaţi-l la finalul scriptului):

- în prima variantă, lucrăm cu răspunsul JSON brut:
    - îl convertim în obiect PHP cu json_decode
    - apoi navigăm prin structura JSON până la vectorul bindings;
    - parcurgem vectorul bindings şi din fiecare înregistrare preluăm cele două valori (URI al subiectului şi numele); avem nevoie de value pentru a accesa valoarea efectivă!
- În a doua variantă, putem lucra ceva mai eficient pe structura generată cu EasyRdf_Sparql_Result:
    - nu mai suntem nevoiţi să studiem structura JSON pentru a naviga prin ea! avem direct un vector de înregistrări la dispoziţie;
    - nu mai e nevoie să extragem valorile cu value, numele proprietăţilor sunt suficiente.

```
print "<br/><br/><b>Afisam lista subiectelor si numelor parcurgand raspunsul JSON prin metode JSON clasice: </b>";
$date=json_decode($rezultatJSON)->results->bindings;
foreach ($date as $inregistrare)
	{
	print "<li>Subiectul ".$inregistrare->subiect->value." are numele ".$inregistrare->nume->value."</li>";
	}

print "<br/><br/><b>Afisam lista subiectelor si numelor parcurgand raspunsul restructurat prin metode EasyRDF: </b><br/>";
foreach ($rezultatRestructurat as $inregistrare)
	{
	print "<li>Subiectul ".$inregistrare->subiect." are numele ".$inregistrare->nume."</li>";
	}
```

În varianta următoare folosim aceeaşi tehnică pentru a realiza crearea unui graf nou, cu identificatorul :grafNou (în acest moment în baza grafuri ar trebui să aveţi doar :grafPersoane şi :grafOrganizatii). Configurările explicite necesare acestei operaţii sunt:

- alipirea, la adresa serviciului ce acceptă operaţii de scriere, a interogării codificate în parametrul update;
- declararea metodei POST, necesară oricărei interogări ce se trimite la adresa pentru operaţii de scriere (indiferent că interogarea este INSERT, DELETE etc.);

Scriptul este următorul:

```
<?php
require 'vendor/autoload.php';
$adresa="http://localhost:8080/rdf4j-server/repositories/grafuri/statements?update=";
$interogare=urlencode("prefix : <http://expl.ro#> insert data {graph :grafNou {:Irina :lucreazaLa :PrimariaCluj}}");
$clienthttp=new EasyRdf_Http_Client($adresa.$interogare);
print $clienthttp->request("POST");
?>
```

Verificaţi apariţia grafului în RDF4J (în browser se va vedea doar mesajul de confirmare a operaţiei).

În varianta următoare înlocuim integral conţinutul din :grafNou cu un graf construit de la zero în PHP. Configurările explicite necesare sunt:

- declararea metodei PUT, care, spre deosebire de POST, nu doar trimite date, ci realizează şi o substituire;
- folosirea unei adrese compatibile cu protocolul GraphStore, astfel încât PUT să substituie conţinutul unui graf integral;
- declararea cu ajutorul antetului Content-Type a formatului în care se va trimite conţinutul nou al grafului; folosim aceeaşi funcţie setHeaders() cu care anterior am modificat antetul Accept; formatul în care se trimit datele va fi N-triples, al cărui cod MIME este text/plain;
- în consecinţă, datele ce se trimit trebuie convertite într-un şir de caractere ce respectă sintaxa N-triples; vom folosi serialise() în acest scop, şi funcţia setRawData pentru a include datele convertite în cerere.

```
<?php
require 'vendor/autoload.php';
$graf=new EasyRdf_Graph("http://expl.ro#grafNou");
$prefixe=new EasyRDF_Namespace();
$prefixe->setDefault("http://expl.ro#");
$graf->addResource("Mihaela","foaf:knows","Petru");
$graf->add("Mihaela","foaf:age","22");

// pana aici am construit graful ce trebuie trimis, in continuare configuram cererea

$adresa="http://localhost:8080/rdf4j-server/repositories/grafuri/rdf-graphs/service?graph=".urlencode("http://expl.ro#grafNou");
$clienthttp=new EasyRdf_Http_Client($adresa);
$clienthttp=$clienthttp->setHeaders("Content-Type","text/plain");
$continutDeTrimis=$graf->serialise("ntriples");
$clienthttp->setRawData($continutDeTrimis);
print $clienthttp->request("PUT");
?>
```

Verificaţi înlocuirea grafului în RDF4J (în browser se va vedea doar mesajul de confirmare a operaţiei).

6. Reprezentarea și procesarea grafurilor prin cod HTML distilabil

6.1 Sintaxa JSON-LD

JSON-LD este o altă modalitate populară de a descrie grafuri de cunoștințe direct în codul sursă HTML, care este însă încorporată în secțiunea JavaScript. Este preferată de Google deoarece permite mai ușor extragerea cunoștințelor și separarea acestora de restul codului HTML.

Să presupunem că dorim să exprimăm în JSON-LD următoarele două grafuri:

```
@prefix x: <http://expl.at#>.
@prefix o: <http://other.at#>.

x:Graph1
{
x:John a x:Human; x:isBrotherOf x:George, x:Anna; x:fullName "John Smith".
x:Anna a x:Human.
x:George a x:Human.
}
o:Graph2
{o:Mary o:worksAt [a o:Company; o:locatedIn o:Wien]}
```

În limbaj natural ar fi: John is a human, is the brother of George and Anna, his full name is "John Smith". Anna is also a human, George is also a human. Mary works at a company located in Wien.

Versiunea JSON-LD pentru acestea ar fi:

```
[{"@context": {"x":"http://expl.at#"},
 "@id":"x:Graph1",
 "@graph":
            [
  {"@id":"x:John","@type":"x:Human",
   "x:isBrotherOf":[{"@id":"x:George","@type":"x:Human"},{"@id":"x:Anna","@type":"x:Human"}],
            "x:fullName":"John Smith"}
   ]},
{"@context": {"o":"http://other.at#"},
 "@id":"o:Graph2",
 "@graph":
            [
  {"@id":"o:Mary",
   "o:worksAt":{"@type":"o:Company","o:locatedIn":{"@id":"o:Wien"}}}
  ]
}
]
```

Explicații:

- întreaga structură conține două secțiuni corespunzătoare celor două grafuri
- fiecare secțiune începe cu @context (unde se declară prefixul), urmat de @id (identificatorul URI al grafului) și secțiunea @graph (conținutul grafului păstrat ca vector)
- în prima secțiune @graph se poate observa:
  - @id creează subiectul (John) urmat de @type dacă dorim să declarăm și clasa subiectului respectiv (Human)
  - o listă cu proprietățile pentru acel subiect (isBrotherOf, fullName)
    - pentru relația brother avem mai multe obiecte grupate într-un vector (George, Ana); fiecare dintre acestea de asemenea primește un anumit tip @type (Human).
    - pentru numele întreg se face doar atribuirea unui string astfel că nu avem nevoie de @id și nici de prefix
- în al doilea @graph avem:
  - un @id ce indică subiectul (Mary) de această dată fără un anume tip

- o Mary este în relație worksAt cu ceva ce nu are @id dar are tip @type – se va crea un nod anonim care este de tip Company; acest nod anonim are locația Viena - Wien (ce are @id și astfel va fi considerat un URI, nu un string) .

Puteți testa codul JSON-LD aici:
http://json-ld.org/playground/

Nu se poate converti în Turtle/TriG, dar se poate vedea versiunea în N-quads pentru a verifica dacă sunt prezente toate afirmațiile solicitate:

JSON-LD Input

```
[{"@context": {"x":"http://expl.at#"},
 "@id":"x:Graph1",
 "@graph":
    [
   {"@id":"x:John","@type":"x:Human",
    "x:isBrotherOf":[{"@id":"x:George","@type":"x:Human"},{"@id":"x:Anna","@type":"x:Human"}],
    "x:fullName":"John Smith"}
    ]},
{"@context": {"o":"http://other.at#"},
 "@id":"o:Graph2",
 "@graph":
    [
   {"@id":"o:Mary",
    "o:worksAt":{"@type":"o:Company","o:locatedIn":{"@id":"o:Wien"}}}
   ]
}
]
```

Expanded | Compacted | Flattened | Framed | N-Quads | Normalized | Signed

```
<http://expl.at#Anna> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://expl.at#Human> <http://expl.at#Graph1> .
<http://expl.at#George> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://expl.at#Human> <http://expl.at#Graph1> .
<http://expl.at#John> <http://expl.at#fullName> "John Smith" <http://expl.at#Graph1> .
<http://expl.at#John> <http://expl.at#isBrotherOf> <http://expl.at#Anna> <http://expl.at#Graph1> .
<http://expl.at#John> <http://expl.at#isBrotherOf> <http://expl.at#George> <http://expl.at#Graph1> .
<http://expl.at#John> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://expl.at#Human> <http://expl.at#Graph1> .
<http://other.at#Mary> <http://other.at#worksAt> _:b0 <http://other.at#Graph2> .
_:b0 <http://other.at#locatedIn> <http://other.at#Wien> <http://other.at#Graph2> .
_:b0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://other.at#Company> <http://other.at#Graph2> .
```

Figura 33 Instrumentul online pentrua sintaxa JSON-LD

Vom face același lucru și pentru următorul exemplu, de această dată cu un singur graf:

```
@prefix x: <http://expl.at#>.
x:Graph1
{
x:Mary x:worksAt x: ABC; x:motherOf x:Peter, x:Andrew; x:hasAge 40 .
x: ABC a x:Company; x:locatedIn x:Wien .
x:Andrew x:isBrotherOf x:Peter; x:hasAge 20 .}
```

Versiunea în limbaj natural este: Mary works at ABC, is the mother of Peter and Andrew, has the age of 40. ABC is a company located in Wien. Andrew is the brother of Peter and has the age of 20.
Aici avem graful în JSON-LD:

```
{
"@context":{"x":"http://expl.at#"},
"@id":"x:Graph1",
"@graph":
[
{"@id":"x:Mary",
  "x:worksAt":{"@id":"x:ABC","@type":"x:Company","x:locatedIn":{"@id":"x:Wien"}},
  "x:motherOf":[{"@id":"x:Peter"},{"@id":"x:Andrew"}],
  "x:hasAge":40},
{"@id":"x:Andrew","x:isBrotherOf":{"@id":"x:Peter"},"x:hasAge":20}
]
}
```

Observați că avem o singură secțiune cu @context/@id/@graph. Graful conține afirmații despre două subiecte (cele două @id din cadrul grafului) și nici unul dintre acestea nu are un tip @type.

6.2 Cerințele Google privind grafuri încorporate

Un alt instrument important pentru testarea grafurilor de cunoștințe încorporate este oferit de Google (care le denumește date structurate - "structured data"):
https://search.google.com/structured-data/testing-tool?hl=ro

Acest instrument nu este un verificator/convertor de sintaxă – poate verifica dacă graful încorporat poate fi înțeles și folosit de motorul de căutare Google (afișarea de Rich Snippets, Knowledge Panel etc.):

- va afișa în partea dreaptă cunoștințele detectate ca un arbore structurat (nu va realiza conversia în alte sintaxe);
- va verifica dacă parantezele sau etichetele sunt corect încadrate
- va verifica dacă există un tip declarat (a se observa mesajul "tip nespecificat" pentru toate elementele din acest exemplu
- dacă se folosește vocabularul Schema.org, va verifica prezența unor proprietăți cheie care sunt solicitate de Google Rich Snippets sau Knowledge Panel pentru a afișa rezultatele căutărilor
- instrumentul poate verifica nu doar sintaxă JSON-LD ci și microdata sau RDFa

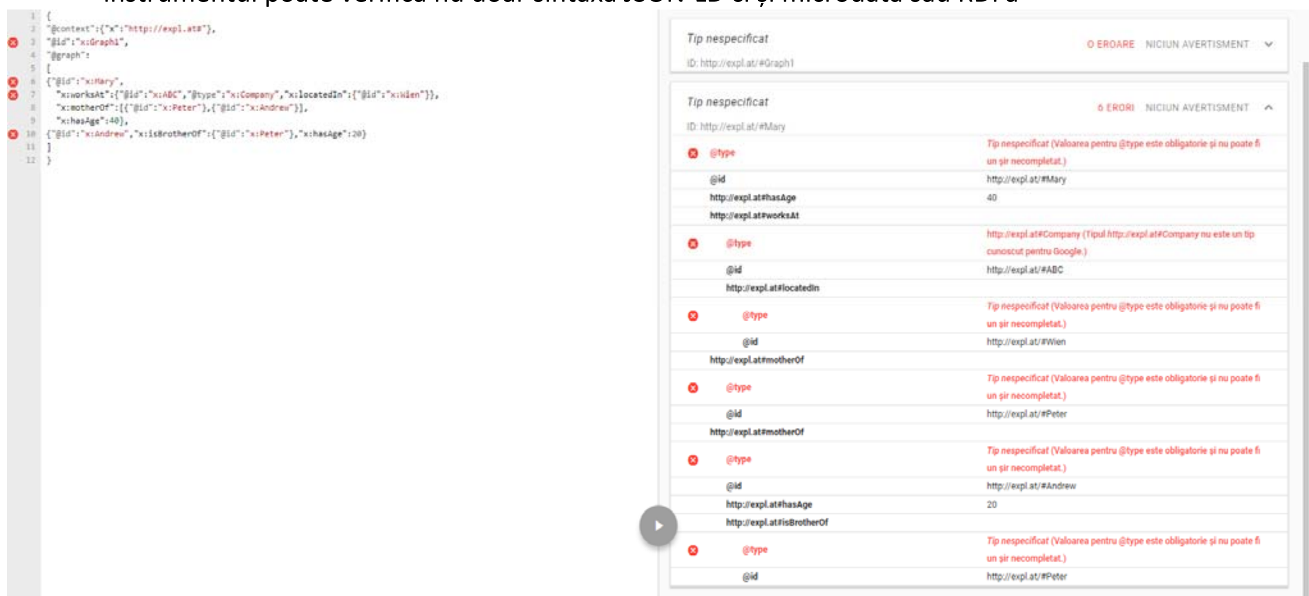


Figura 34 Instrumentul online oferit de Google pentru date structurate – verificarea sintaxei JSON-LD

În acest exemplu nu am folosit concepte schema.org. Verificați următorul cod HTML cu proprietăți microdata[53] și schema.org:

```
<div itemscope="itemscope" itemid="http://expl.at/Avatar" itemtype ="http://schema.org/Movie" >
<h1 itemprop="name">Avatar</h1>
Directed by:
        <span itemprop="director" itemid="http://expl.at/JamesCameron"
                itemscope="itemscope" itemtype="http://schema.org/Person" >
                <span itemprop="name">James Cameron</span>
                <meta itemprop="birthDate" content="August 16, 1954"/>
                 <link itemprop="url" href="https://en.wikipedia.org/wiki/James_Cameron"/>
                <span itemprop="spouse" itemscope="itemscope" itemid="http://expl.at/SuzyAmis"
                                itemtype="http://schema.org/Person"/>
        </span>
</div>
```

[53] https://www.w3.org/TR/microdata/

```
<div itemscope="itemscope" itemid="http://expl.at/Avatar" itemtype ="http://schema.org/Movie" >
<h1 itemprop="name">Avatar</h1>
Directed by:
    <span itemprop="director" itemid="http://expl.at/JamesCameron"
        itemscope="itemscope" itemtype="http://schema.org/Person" >
        <span itemprop="name">James Cameron</span>
        <meta itemprop="birthDate" content="August 16, 1954"/>
         <link itemprop="url" href="https://en.wikipedia.org/wiki/James_Cameron"/>
        <span itemprop="spouse" itemscope="itemscope" itemid="http://expl.at/SuzyAmis"
                itemtype="http://schema.org/Person"/>
    </span>
</div>
```

Movie All (1)

Movie O EROARE UN AVERTISMENT

ID: http://expl.at/Avatar

@type	Movie
@id	http://expl.at/Avatar
name	Avatar
director	
@type	Person
@id	http://expl.at/JamesCameron
name	James Cameron
birthDate	1954-08-16
url	https://en.wikipedia.org/wiki/James_Cameron
spouse	
@type	Person
@id	http://expl.at/SuzyAmis
image	Este obligatorie o valoare pentru câmpul *image*.
dateCreated	Câmpul *dateCreated* este recomandat. Indicați o valoare, dacă este disponibilă.

Figura 35 Instrumentul online oferit de Google pentru date structurate - verificare cod microdata

Instrumentul va raporta:

- Google a determinat că avem o porțiune de cod HTML unde descriem o instanță Movie și o instanță Person (clase/tipuri stabilite de schema.org);
- pentru aceste tipuri cunoscute, Google urmărește să găsească anumite informații pe care să le folosească la afișarea rezultatelor de căutare îmbunătățite (Rich Snippets, Rich Cards etc.). Pentru Movie, se așteaptă să indicăm și o imagine reprezentând filmul (proprietatea image, obligatorie) și data creării dateCreated (opțional). Acestea sunt solicitate pentru Movie (și nu pentru Person) deoarece eroarea este afișată aliniată cu tipul Movie.

Putem consulta pagina schema.org pentru Movie pentru a vedea o listă completă a proprietăților prin care putem descrie Movies:
https://schema.org/Movie

Totuși, nu toate sunt considerate esențiale pentru Google Rich Snippets. Puteți verifica lista cu proprietățile de care Google este „interesat” la adresa următoare:
https://developers.google.com/search/docs/data-types/tv-movies
(verificați meniul din partea stângă pentru a citi mai multe și despre alte lucruri pe lângă Movies – se preconizează o evoluție în viitor și motoarele de căutare vor urmări să găsească din ce în ce mai multe proprietăți pentru diverse lucruri pe care oamenii le caută).

Putem consulta detalii despre proprietatea image de la schema.org:
https://schema.org/image

Observați că are unele subproprietăți (logo, photo), un domeniu (Thing) și trebuie să specifice un obiect imagine sau un fișier (codomeniul este mulțimea dată de uniunea dintre URL și ImageObject).
Deci, dacă dorim să scăpăm de eroarea arătată de instrumentul Google trebuie să includem în itemscope o anumită imagine pentru instanța Movie (de ex. în cadrul elementului DIV ce specifică filmul Avatar), și să o marcăm cu proprietatea itemprop=”image”.

```
<div itemscope="itemscope" itemid="http://expl.at/Avatar" itemtype ="http://schema.org/Movie" >
<img itemprop="image" src="myimgfile.png"/>
<h1 itemprop="name">Avatar</h1>
Directed by:
        <span itemprop="director" itemid="http://expl.at/JamesCameron"
                itemscope="itemscope" itemtype="http://schema.org/Person" >
                <span itemprop="name">James Cameron</span>
                <meta itemprop="birthDate" content="August 16, 1954"/>
    <           meta itemprop="image" src="f.png"/>
                <link itemprop="url" href="https://en.wikipedia.org/wiki/James_Cameron"/>
```

```
			<span itemprop="spouse" itemscope="itemscope" itemid="http://expl.at/SuzyAmis"
				itemtype="http://schema.org/Person"/>
		</span>
</div>
```

Dacă nu dorim ca imaginea să fie vizibilă pe pagină și este inclusă aici pentru referință doar pentru a satisface cerințele Google (acelea de a putea fi afișată în rezultatele îmbunătățite de căutare) putem să o ascundem cu CSS sau putem să folosim o etichetă LINK în loc de IMG:

```
<link itemprop="image" href="myimgfile.png"/>
```

Observați poziția în cadrul codului – este așezată între DIV -ul care are Avatar ca itemid (și un itemscop declarat) dar NU în cadrul etichetei SPAN care are pe JamesCameron în itemid (de asemenea cu un itemscop declarat). Acest lucru clarifică faptul că imaginea este o proprietate a elementului Avatar și nu o proprietate a lui JamesCameron.

6.3 Sintaxa HTML microdata și terminologia Schema.org

În următorul exercițiu, vom crea o pagină HTML cu microdata care va afișa informații despre două evenimente, într-o modalitate ce satisface cerințele Google pentru marcare semantică. În plus, vom extrage și procesa aceste cunoștințe pe partea de client (JavaScript) pentru a beneficia de anumite funcționalități suplimentare – o hartă Google pe care vor fi însemnate locațiile evenimentelor în funcție de informațiile disponibile în fragmentele microdata procesate:

În primul rând vom crea pagina fără microdata, doar cu informațiile pe care le dorim vizibile în browser, ce anunță două evenimente. Pentru fiecare eveniment vom indica:

- un link către un website ce descrie evenimentul
- adresa unde va avea loc evenimentul
- o imagine care va arăta adresa (de ex. sigla unui club – pregătiți 2 imagini PNG pentru 2 cluburi și includeți-le în cod cu IMG așa cum este arătat)
- data și ora evenimentului

```
<html>
<body>
  <h1>Future Events</h1>

  <div>
		<h2><a href="http://themadonnaeventpage.com">Madonna Concert</a></h2>
		<p> You will have fun at the concert of Madonna </p>
		in Club Bamboo,Strada Tuzla 50,Bucharest,Romania<br/>
		<img src="Imagine1.png" height="100" width="100"/>
		<div>November 2, 9:00pm - 11:00pm</div>
  </div>

  <div>
		<h2><a href="http://themetallicaeventpage.com">Metallica Concert</a></h2>
		<p>You will have fun at the concert of Metallica</p>
		in NOA Club,Strada Republicii 108,Cluj-Napoca,Romania<br/>
		<img src="Imagine2.png" height="100" width="100"/>
		<div>October 13, 7:00pm - 11:00pm</div>
  </div>

</body>
</html>
```

Ar trebui să arate în felul următor în browser:

Future Events

Madonna Concert

You will have fun at the concert of Madonna

in Club Bamboo,Strada Tuzla 50,Bucharest,Romania

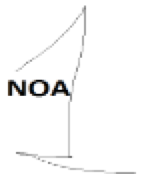


November 2, 9:00pm - 11:00pm

Metallica Concert

You will have fun at the concert of Metallica

in NOA Club,Strada Republicii 108,Cluj-Napoca,Romania



October 13, 7:00pm - 11:00pm

Figura 36 Vizualizarea paginii în browser

În felul în care este, pagina poate fi înțeleasă de un om, dar nu de un client software sau de un motor de căutare. Google este în măsură să caute șiruri în conținutul HTML, dar nu va putea să înțeleagă că Madona este cântăreață (și nu o statuie de exemplu), că Bamboo este o locație (și nu o plantă) sau că 9 reprezintă ora de începere a evenimentului (și nu doar o valoare întreagă). Pentru a face posibil acest lucru este nevoie să încorporăm cunoștințe semantice.

În primul rând este nevoie să declarăm care sunt subiecții principali – entitățile despre care dorim să facem afirmații. Mai exact, este nevoie:

- să declarăm care este tipul acestora, în corespondență cu schema.org (în cazul nostru MusicEvent)
- să creăm pentru fiecare entitate câte un itemscope (evenimentul). Un itemscope este un element HTML (în cazul nostru un atribut în elementul DIV) ce va conține toate informațiile (atât conținutul vizibil cât și afirmațiile încorporate) despre un anumit subiect. Elementul DIV va avea nevoie de următoarele atribute:
    - ITEMSCOPE (pentru a defini acea porțiune din pagină care este dedicată subiectului)
    - ITEMTYPE (unde declarăm tipul pentru fiecare subiect)
    - și opțional ITEMID (dacă dorim să dăm fiecărui subiect un URI pentru a permite altora să îl refolosească pe viitor sau dacă știm că subiectul are un URI popular – vom omite acest pas deoarece nu suntem interesați să ne conectăm afirmațiile cu alte baze de cunoștințe; mai mult, evenimentele sunt temporare și cel mai probabil nu vom găsi cunoștințe despre acestea înainte de a se realiza)

Pentru fiecare eveniment este nevoie să includem aceste atribute după cum se vede:

```
<html>
<body>
  <h1>Future Events</h1>

  <div itemscope="itemscope" itemtype="http://schema.org/MusicEvent">
                    <h2><a href="http://themadonnaeventpage.com">Madonna Concert</a></h2>
......
  </div>

  <div itemscope="itemscope" itemtype="http://schema.org/MusicEvent">
                    <h2><a href="http://themetallicaeventpage.com">Metallica Concert</a></h2>
.......
  </div>

</body>
</html>
```

Identificatorul URI a tipului/clasei MusicEvent a fost luat de pe site-ul schema.org – este o subclasă a clasei Event (care la rândul ei este o subclasă a clasei Thing).

La pasul următor, dorim să selectăm cele mai relevante proprietăți (cel puțin pentru Google) care descriu evenimentul muzical. Putem să ne uităm pe site-ul schema.org, însă lista afișată este foarte lungă. Metoda preferată este de a selecta cele importante pentru Google. Pentru aceasta, vom testa codul nostru în instrumentul de testare Google:

https://search.google.com/structured-data/testing-tool

Ar trebui să vedem câteva dintre proprietățile pe care Google le consideră importante și pe care nu le găsește în codul curent. Vom include proprietățile ce sunt cu roșu (obligatorii) și câteva dintre cele portocalii.

MusicEvent	3 ERRORS 6 WARNINGS
@type	MusicEvent
location	A value for the *location* field is required.
name	A value for the *name* field is required.
startDate	A value for the *startDate* field is required.
description	The *description* field is recommended. Please provide a value if available.
endDate	The *endDate* field is recommended. Please provide a value if available.
image	The *image* field is recommended. Please provide a value if available.
offers	The *offers* field is recommended. Please provide a value if available.
performer	The *performer* field is recommended. Please provide a value if available.
url	The *url* field is recommended. Please provide a value if available.

Figura 37 Proprietățile considerate de Google relevante pentru a fi incluse în codul HTML

Pentru a include aceste proprietăți:

- trebuie să decidem dacă dorim ca valorile acestora să fie vizibile în browser pentru un utilizator uman (sau doar pentru clienții software/motoarele de căutare); deoarece avem deja informații vizibile (datele, locurile etc.) le vom lăsa vizibile
- trebuie să verificăm la adresa http://schema.org/MusicEvent care este codomeniul (tipul așteptat) pentru fiecare proprietate, deoarece acesta influențează modul în care îl includem în HTML:
    - name (numele evenimentului) are codomeniul Text, astfel putem să îl specificăm cu o simplă etichetă SPAN care marchează numele evenimentului (deja vizibil);
    - url (pagina oficială a evenimentului) are codomeniul URL, astfel putem să îl specificăm ca un hiperlink către pagina oficială a evenimentului (fictiv);
    - performer (cel care realizează evenimentul) are codomeniu Organization sau Person astfel trebuie să indice un lucru (thing); acesta trebuie să aibă propriul itemscope/itemtype plus câteva alte proprietăți, de aceea vom crea un element SPAN;
    - location (unde are loc evenimentul) poate fi un text simplu dar poate fi și ceva de tip Place; în al doilea caz este nevoie să aibă propriul itemscope/itemtype plus alte proprietăți proprii și de aceea vom crea un element DIV pentru locație
    - startDate trebuie să fie o valoare date-time care poate fi definită în HTML 5 cu TIME; același lucru este valabil și pentru endDate.

Urmăriți cum includem toate acestea în cadrul itemscope al fiecărui eveniment:

```
<html>
<head>
</head>
<body>
<h1>Future Events</h1>

<div itemscope="itemscope" itemtype="http://schema.org/MusicEvent">
        <h2><a href="http://themadonnaeventpage.com" itemprop="url"><span itemprop="name">Madonna Concert</span></a></h2>
        <p> You will have fun at the concert of
                <span itemprop="performer" itemtype="http://schema.org/MusicGroup" itemscope="itemscope">Madonna</span>
        </p>
         in
        <div itemprop="location" itemtype="http://schema.org/Place" itemscope="itemscope">
                Club Bamboo,Strada Tuzla 50,Bucharest,Romania<br/>
                <img src="Imagine1.png" height="100" width="100"/>
        </div>
        <div>
                <time datetime="2017-11-02T21:00:00+02:00" itemprop="startDate">November 2, 9:00pm</time> -
                <time datetime="2017-11-02T23:00:00+02:00" itemprop="endDate">11:00pm</time>
        </div>
</div>

<div itemscope="itemscope" itemtype="http://schema.org/MusicEvent">
        <h2><a href="http://themetallicaeventpage.com" itemprop="url"><span itemprop="name">Metallica Concert</span></a></h2>
        <p>You will have fun at the concert of
                 <span itemprop="performer" itemtype="http://schema.org/MusicGroup" itemscope="itemscope">Metallica</span>
         </p>
         in
        <div itemprop="location" itemtype="http://schema.org/Place" itemscope="itemscope">
                NOA Club,Strada Republicii 108,Cluj-Napoca,Romania<br/>
                <img src="Imagine2.png" height="100" width="100"/>
         </div>
        <div>
                <time datetime="2017-10-13T19:00:00+02:00" itemprop="startDate">October 13, 7:00pm</time> -
                <time datetime="2017-10-13T23:00:00+02:00" itemprop="endDate">11:00pm</time>
        </div>
</div>

</body>
</html>
```

Câteva observații suplimentare:

- datele sunt scrise în format standardizat, incluzând +2:00 diferența fus orar GMT
- pentru proprietatea performer, nu am atribuit nici o Persoană sau Organizație după cum se indică pe schema.org; în schimb, am căutat în subclasele de la Organization și am găsit o clasă mai potrivită pentru exemplul nostru: MusicGroup (ce cuprinde și cântăreți individuali);
- am adoptat abordarea de a folosi toate proprietățile vizibile în browser; dacă dorim să includem proprietăți care nu trebuie să fie afișate în browser atunci trebuie să folosim alte etichete:
  - LINK pentru tot ce face referire la un fișier/url (în loc de A HREF sau IMG SRC);
  - META pentru proprietățile care au o valoare simplă, cu atributul CONTENT pentru a păstra această valoare (în loc de DIV/SPAN).

Dacă testăm acum codul nostru în instrumentul Google încă mai găsim câteva erori/atenționări. Acest lucru se datorează faptului că am inclus două tipuri noi de lucruri – MusicGroup și Place! Fiecare dintre acestea trebuie să fie descrise folosind aceeași strategie. După verificarea proprietăților relevante, hotărâm să includem următoarele (în funcție și de ce informație este deja vizibilă în pagină):

- pentru MusicGroup:
  - numele cu un simplu string ca valoare
- pentru Place:
  - numele cu un simplu string ca valoare
  - proprietatea image, având deja disponibile imaginile cu cluburile respective
  - proprietatea address, a cărei codomeniu este clasa PostalAddress sau text; am ales să o includem ca un obiect de tip PostalAddress, deoarece ne dă acces detaliat către fiecare componentă a adresei (stradă, oraș etc.). De aceea vom crea un nou itemscope de tip PostalAddress cu următoarele proprietăți – toate având valori text simple:
    - streetAddress
    - addressLocality
    - addressCountry

```
<html>
<head>
</head>
<body>
        <h1>Future Events</h1>

        <div itemscope="itemscope" itemtype="http://schema.org/MusicEvent">
        <h2><a href="http://themadonnaeventpage.com" itemprop="url"><span itemprop="name">Madonna Concert</span></a></h2>

        <p> You will have fun at the concert of
                <span itemprop="performer" itemtype="http://schema.org/MusicGroup" itemscope="itemscope">
                        <span itemprop="name">Madonna</span>
                </span>
        </p>
        in
        <div itemprop="location" itemtype="http://schema.org/Place" itemscope="itemscope">
                <span itemprop="name">Club Bamboo</span>,
                <span itemprop="address" itemtype="http://schema.org/PostalAddress" itemscope="itemscope">
                        <span itemprop="streetAddress">Strada Tuzla 50</span>,
                        <span itemprop="addressLocality">Bucharest</span>,
                        <span itemprop="addressCountry">Romania</span>
                </span>
                <br/>
                <img itemprop="image" src="Imagine1.png" height="100" width="100"/>
        </div>
        <div>
                <time datetime="2017-11-02T21:00:00+02:00" itemprop="startDate">November 2, 9:00pm</time> -
                <time datetime="2017-11-02T23:00:00+02:00" itemprop="endDate">11:00pm</time>
        </div>
        </div>

        <div itemscope="itemscope" itemtype="http://schema.org/MusicEvent">
        <h2><a href="http://themetallicaeventpage.com" itemprop="url"><span itemprop="name">Metallica Concert</span></a></h2>
```

```
            <p>You will have fun at the concert of
                  <span itemprop="performer" itemtype="http://schema.org/MusicGroup" itemscope="itemscope">
                        <span itemprop="name">Metallica</span>
                  </span>
            </p>
            in
            <div itemprop="location" itemtype="http://schema.org/Place" itemscope="itemscope">
                  <span itemprop="name">NOA Club</span>,
                  <span itemprop="address" itemtype="http://schema.org/PostalAddress" itemscope="itemscope">
                        <span itemprop="streetAddress">Strada Republicii 108</span>,
                        <span itemprop="addressLocality">Cluj-Napoca</span>,
                        <span itemprop="addressCountry">Romania</span>
                  </span>
                  <br/>
                  <img itemprop="image" src="Imagine2.png" height="100" width="100"/>
            </div>
            <div>
                  <time datetime="2017-10-13T19:00:00+02:00" itemprop="startDate">October 13, 7:00pm</time> -
                  <time datetime="2017-10-13T23:00:00+02:00" itemprop="endDate">11:00pm</time>
            </div>
      </div>

</body>
</html>
```

A se observa părțile îngroșate, care reprezintă noile secțiuni incluse. Dacă testăm acum codul în instrumentul de testare Google nu vor mai fi erori, doar anumite atenționări pentru câteva recomandări de proprietăți suplimentare pe care le vom ignora. Observați că în browser nu s-a modificat nimic – toate aceste informații sunt cunoștințe machine-readable pentru clienții care pot să înțeleagă codul sursă sau pentru motoare de căutare care vor indexa anunțurile despre evenimentele noastre!

Vom crea o pagină client care va extrage cunoștințe din această pagină și le va folosi pentru a afișa locațiile evenimentelor pe o hartă Google Map. Pentru aceasta vom crea o a doua pagină HTML care va folosi JavaScript pentru a extrage conținutul paginii și pentru a crea o harta Google Map. Mai mult, pentru simplitate, în loc de cod JavaScript vom folosi frameworkul JQuery:

În acest scop, trebuie să învățăm următoarele:

- cum să preluăm conținutul unei alte pagini și să extragem cunoștințele pe care le găsim acolo;
- cum să creăm o hartă GoogleMap și cum șă îi transmitem informații procesate.

6.4 Distilarea microdata cu JavaScript

În primul rând, deoarece vom lucra cu cereri AJAX trebuie să găzduim paginile pe un server web (XAMPP/Apache, în directorul htdocs) și să le accesăm prin http://localhost.

În al doilea rând, e mai bine să folosim o bibliotecă JavaScript existentă pentru extragerea cunoștințelor. Putem să folosim și cod JavaScript nativ cu getElementBy... însă efortul de a verifica fiecare itemprop din fiecare itemscope poate fi evitat dacă folosim biblioteci existente. O astfel de bibliotecă este disponibilă la adresa:
https://github.com/hubgit/jquery-microdata

Aceasta e o bibliotecă construită peste JQuery și este deja inclusă. Oricum, avem nevoie de JQuery pentru a executa cereri AJAX și astfel le putem folosi pe ambele. După ce descărcați biblioteca, căutați în arhivă următoarele două fișiere:

- Jquery.js – versiunea jquery pe baza căreia a fost realizat distilatorul (distiller)
- jquery.microdata.js -distilatorul propriu-zis

Copiați ambele fișiere în folderul htdocs a serverului Apache

De asemenea, copiați exemplul creat anterior cu numele microdatasource.html. Asigurați-vă că îl puteți vedea la adresa http://localhost/microdatasource.html.

Vom crea următorul exemplu în același folder cu numele microdataconsumer.html.

```
<html>
<head>

<script src="jquery.js"></script>
<script src="jquery.microdata.js"></script>
<script type="text/javascript">
function initializare()
            {
            $.get("microdatasource.html",procesarePagina)
            }

function procesarePagina(raspuns)
            {
            alert(raspuns)
            arboreDOM=$("<div/>").append(raspuns)
            alert(arboreDOM.find("h1").text())
            }

</script>

</head>
<body onload="initializare()">
 Aceasta este pagina client
</body>
 </html>
```

Observații:

- observați cele două script-uri care importă JQuery și distilatorul din folderul curent (asigurați-vă că cele două fișiere sunt corect plasate);
- secțiunea body este goală, doar apelează funcția responsabilă cu extragerea întregului conținut din fișierul creat anterior – microdatasource.html;
- funcția realizează cereri AJAX HTTP folosind metoda GET. În JavaScript trebuie să facem un XMLHttpRequest, însă în JQuery este mai ușor deoarece avem funcția $.get – trebuie doar să specificăm care ar fi fișierul accesat și funcția care ar procesa răspunsul după ce îl primim;
    - Această funcție este procesarePagina și implicit are un argument care conține răspunsul. Trebuie să facem doar câteva operații simple pentru a verifica dacă totul funcționează corect:
        - pentru început afișăm întregul răspuns – ar trebui să puteți vedea un mesaj de alertă cu întreg codul HTML al fișierului accesat;
        - apoi, trebuie să convertim răspunsul (care este un string) într-un arbore DOM obiectual care poate fi procesat de un JQuery (nu este posibil să executăm funcții JQuery direct pe un string, cele mai multe solicită un arbore DOM complet sau un set de noduri DOM!) Cea mai ușoară modalitate de a realiza acest lucru în JQuery este:
            - să creăm în memorie un element rădăcină DIV gol; pentru aceasta folosim funcția $ din JQuery, care creează în memorie un element dacă argumentul acesteia este o etichetă HTML;
            - să completăm acest element DIV din memorie cu toate elementele (arborele DOM) pe care le găsim în răspunsul string; pentru aceasta folosim funcția append, care convertește un string în noduri DOM și le include în cadrul elementului care a apelat funcția; observați că am fi putut insera codul HTML primit în cadrul elementului BODY, dar acest lucru l-ar fi făcut vizibil în browser și nu este ceea ce am dori – vrem doar să procesăm microdatele

- vom testa apoi această conversie dacă s-a realizat cu succes prin căutarea elementului H1 (cu funcția find() aplicată pe arborele DOM) și afișarea valorii sale (cu funcția text() care ne va da “Future Events”)

Executați exemplul prin localhost (nu prin simplu dublu-click – chiar dacă este un fișier HTML simplu, cererea AJAX solicită acces prin HTTP!) Ar trebui să puteți vedea 2 mesaje de alertă – primul este întregul cod HTML din primul fișier, apoi mesajul “Future Events” demonstrând faptul că fișierul curent poate să extragă informații din primul fișier (preluat prin AJAX).

Dacă totul decurge bine, trebuie să adăugăm funcția care extrage microdate și afișează ceva din acestea, pentru a demonstra că funcționează.

```
<html>
<head>

<script src="jquery.js"></script>
<script src="jquery.microdata.js"></script>
<script type="text/javascript">
function initializare()
        {
        $.get("microdatasource.html",procesarePagina)
        }

function procesarePagina(raspuns)
        {
        arboreDOM=$("<div/>").append(raspuns)
        evenimente=arboreDOM.items("http://schema.org/MusicEvent")
        adrese=evenimente.map(construiesteAdrese)
        alert(adrese[0]+adrese[1])
        }


function construiesteAdrese()
        {
        numeClub=$(this).property("location").property("name").value()
        obiectAdresa=$(this).property("location").property("address")
        stradaClub=obiectAdresa.microdata("streetAddress")
        localitateClub=obiectAdresa.microdata("addressLocality")
        taraClub=obiectAdresa.microdata("addressCountry")
        adresaClub=numeClub+", "+stradaClub+", "+localitateClub+", "+taraClub
        return adresaClub
        }

</script>

</head>
<body onload="initializare()">
Aceasta este pagina client
</body>
    </html>
```

Observații:

- după ce am creat arborele DOM cu tot codul HTML obținut din celălalt fișier, apelăm funcția items(). Aceasta este oferită de biblioteca distilatorului – preia ca argument identificatorul URI a unei clase/ tip (în cazul de față MusicEvent) și returnează un vector (evenimente) cu toate subiectele care au fost identificate în acel arbore. Pentru fiecare subiect (numit și “item”) se vor crea niște funcții speciale care ne vor permite să accesăm proprietățile fără să folosim funcțiile tradiționale JavaScrit/DOM;
- În următoarea linie de cod, apelăm funcția construieșteAdrese pentru fiecare eveniment detectat. Această funcție preia ca argument fiecare eveniment și returnează adresa acestuia ca string, prin extragerea și concatenarea părților din adresă găsite în diverse proprietăți ale microdatelor (numele clubului, strada, localitatea, țara). Vom folosi o funcție specială numită map() (ce este

oferită de JQuery) – în general map() preia ca argument o funcție, ce o va executa în mod repetitiv pe fiecare element al vectorului și construiește un vector nou format din rezultatele date de acele apeluri de funcție. În cazul nostru, funcția care urmează să fie executată în mod repetitiv este construieșteAdrese, vectorul input este evenimente (toate evenimentele detectate) și vectorul output este adrese (care va colecta toate adresele construite de către construiesteAdrese pentru fiecare element MusicEvent).

- Să examinăm construiesteAdrese():
    - o $(this) se referă la evenimentul curent care trebuie procesat (construiesteAdrese este apelată în mod repetat pentru fiecare eveniment);
    - o $(this).property("location").property("name").value() trebuie interpretată în felul următor: pentru fiecare eveniment curent se preia proprietatea locația; valoarea acesteia trebuie să fie ceva ce are o proprietate name; se preia valoarea string a numelui.
    - o $(this).property("location"). property("address") trebuie interpretată astfel: pentru evenimentul curent se preia locația; valoarea acesteia ar trebui să fie ceva care are o proprietate adresa care de asemenea este un anume lucru; se returnează acest lucru ca un obiect (nu am folosit value() de această dată pentru este nevoie să extragem strada/localitatea/țara);
    - o obiectAdresa.microdata("streetAddress") trebuie interpretată în felul următor: pentru un obiect dat se caută o proprietate microdata numită "streetAddress" și returnează valoarea acesteia ca un string (funcția microdata() este o scurtătură pentru property().value() – puteam să o folosim încă de la început);
    - o pentru o documentație completă a funcțiilor distilatorului să se acceseze:

      https://github.com/hubgit/jquery-microdata

      și

      http://hublog.hubmed.org/archives/001979.html
    - o se poate deduce că principiul distilatorului este de a crea un tip nou de arbore (precum arborele DOM) – un arbore microdata care ignoră toate elementele HTML ce nu reprezintă proprietățile microdata și care poate fi navigat ierarhic (de la subiect către proprietățile acestuia și spre alte lucruri conectate de aceste proprietăți). Avem posibilitatea de a înlănțui proprietăți multiple pentru a merge mai adânc în jos pe ramuri și pentru a prelua valorile proprietăților (putem schimba, de asemenea, valorile proprietăților! Verificați documentația pentru detalii cu privire la modul de a scrie microdate în mod dinamic în codul HTML utilizând același distilator)
    - o În cele din urmă, funcția colectează toate componentele unei adrese, le concatenează într-un string și returnează acel string (care urmează să fie păstrate în vectorul adrese).

Executați exemplul prin localhost și ar trebui să vedeți un mesaj de alertă cu două adrese realizate din informația încorporată ca microdata.
Acum, că suntem capabili de a extrage informații din microdata, ar trebui să folosim aceste informații într-un mod semnificativ. De exemplu, pentru a indica adresele într-un GoogleMap.

6.5 Crearea unei hărți GoogleMaps

Dacă doriți ca să includeți o hartă GoogleMap într-un site web trebuie să vă abonați la două servicii API oferite de Google:

- GoogleMaps vă permite să creați și să manipuleze hărți GoogleMap în propria pagină web;
- Geocoder preia ca input o adresă și returnează coordinatele geografice ale sale; acestea sunt necesare dacă se dorește realizarea unei acțiuni cu o anumită locație pe o hartă GoogleMap - de exemplu, pentru a le evidenția cu markeri vizuali.

Ambele servicii pot fi accesate prin intermediul cererilor AJAX.

Puteți beneficia de aceste servicii Google gratuit:

- Dacă aveți un cont Google
- Dacă solicitați o "cheie API" - un cod propriu pe care trebuie să îl includeți în cererile AJAX
- Dacă nu se abuzează de servicii: Google limitează solicitările la câteva mii de cereri AJAX pe zi. Acest lucru este suficient pentru exerciții (trebuie avut grijă să nu se partajeze cheia API cu alte persoane, altfel există riscul ca cererile AJAX gratuite să fie utilizate de alții!). Pentru un site web real ar trebui să se plătească pentru a se permite realizarea mai multor cereri AJAX decât ceea ce este permis prin abonamentul gratuit).
  - Puteți consulta limitările folosirii gratuite pentru serviciul GoogleMaps, precum și avantajele de a plăti pentru upgrade-uri, la https://developers.google.com/maps/documentation/javascript/usage
  - Puteți consulta limitările folosirii gratuite pentru serviciul de Geocoder la https://developers.google.com/maps/documentation/javascript/geocoding#UsageLimits
  - Pentru următoarele exerciții nu ar trebui să plătească, dacă sunt folosite cu măsură
    - asigurați-vă că nimeni nu poate vedea cheia API,
    - asigurați-vă că nu sunt făcute cereri AJAX inutile - atunci când doriți să testați alte părți ale codului care nu utilizează aceste servicii, ar trebui să comentați apelurile AJAX,
    - contul dvs. Google va oferi, de asemenea, unele opțiuni pentru a securiza cheia API-ului (de exemplu, pentru a le permite numai dintr-o singură adresă IP) – deși astfel de măsuri nu ar trebui să fie necesare dacă sunteți atenți cu punctele anterioare.

Pentru a activa accesul la astfel de servicii trebuie să vă conectați la Google Developer Console (cu contul dvs. Google):
https://console.developers.google.com/

Acolo se va cere să se definească un "proiect" (o modalitate de a păstra toate serviciile Google împreună). După aceea, veți putea vedea consola API Manager:

- în Credentials puteți crea și gestiona cheia API (nu utilizați OAuth sau Service account options, doar cheia API, care ar trebui să fie un șir lung pe care trebuie să îl includeți ca un parametru în cererile AJAX)
- În Library puteți căuta și selecta serviciile Google de care aveți nevoie
- În Dashboard ar trebui să vedeți lista de servicii pe care le puteți utiliza (în lista ar trebui să aveți Google Maps JavaScript API și Google Maps Geocoding API - în cazul în care există o opțiune Disable pe partea dreaptă, aceasta înseamnă că serviciile sunt activate și în lucru; aveți posibilitatea să monitorizați de aici cât de mult ați folosit din limita zilnică)

Puteți verifica, de asemenea, etapele prezentate la
https://developers.google.com/maps/documentation/javascript/adding-a-google-map#key

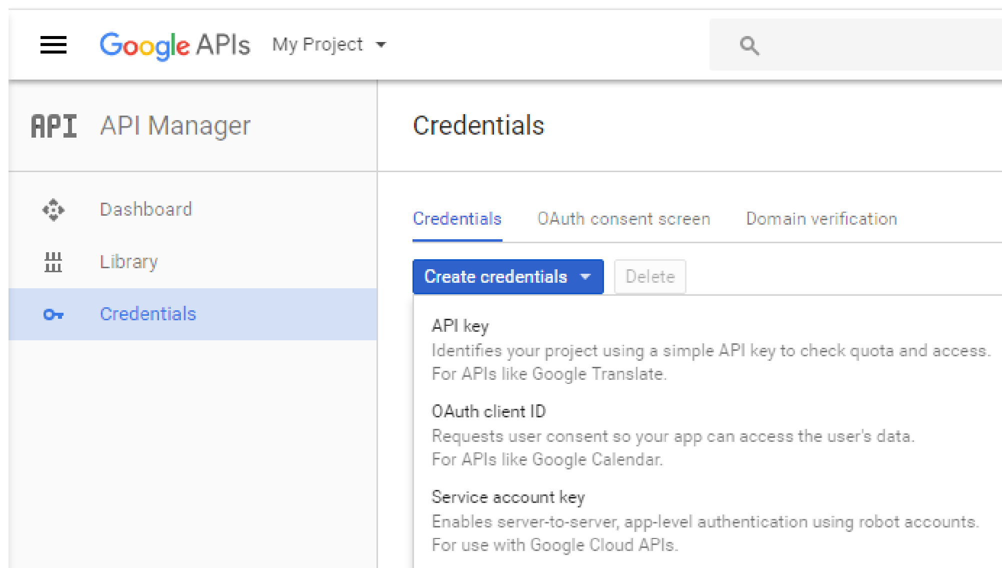


Figura 38 Consola API Manager

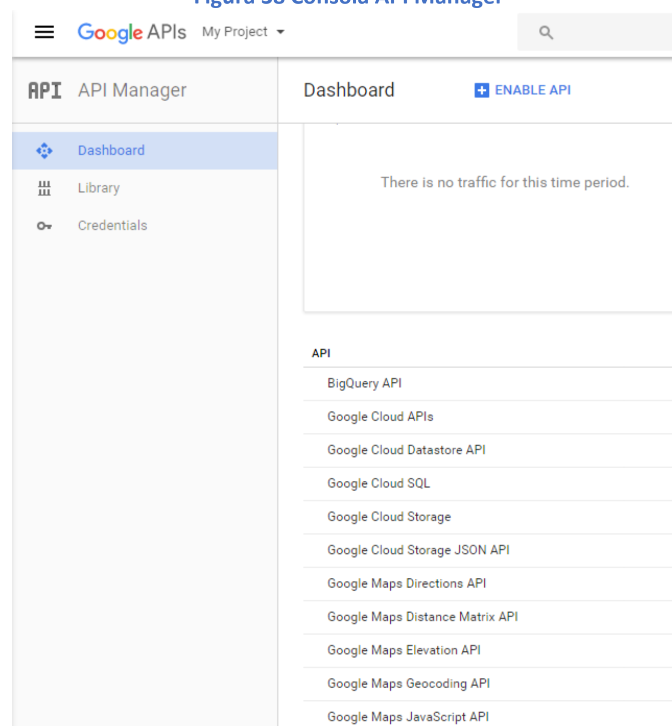


Figura 39 Lista de servicii Google care ar putea fi utilizate

Dacă aveți cheia API, creați o pagină simplă cu o hartă Google:

```
<html>
<head>

<script async defer type="text/javascript" src="//maps.google.com/maps/api/js?key=...&callback=initializareHarta"></script>
<script type="text/javascript">
function initializareHarta()
        {
        spatiuRezervat=document.getElementById("harta")
        configurari={center:{lat:46.773579,lng:23.620014},zoom:8}
        harta=new google.maps.Map(spatiuRezervat,configurari)
        }
</script>
</head>

<body>
        <div style="border:2px solid red;width:500px;height:500px" id="harta"></div>
</body>
```

```
</html>
```

Asigurați-vă că includeți propria dvs. cheie API în loc de puncte. De asemenea, există și alte atribute pe care trebuie să le includeți:

- ASYNC și DEFER pentru a vă asigura că harta este încărcată asincron (în cazul în care se execută altceva pe aceeași pagină, nu ar trebui să se întrerupă funcționalitatea);
- CALLBACK este un parametru ce trebuie specificat pentru a indica funcția care va inițializa harta:
    - unde va fi poziționată aceasta (avem un element DIV gol rezervat pentru asta)
    - În ce poziție va fi centrată și cât de mult se poate mări/micșora.

Puteți verifica, de asemenea, documentația pentru Google Maps API:
https://developers.google.com/maps/documentation/javascript/
(Aceasta este doar pentru JavaScript, pentru aplicațiile mobile există o documentație diferită)

În următorul exercițiu dorim să plasăm un marcator (marker) vizual pe hartă. Nu cunoaștem coordonatele acelui loc, dar avem adresa într-un șir de caractere - avem nevoie de serviciul Geocoder pentru a converti adresa la coordonatele sale. Ar trebui să verificați și documentația Geocoder:
https://developers.google.com/maps/documentation/geocoding/intro

```
<html>
<head>

<script async defer type="text/javascript" src="//maps.google.com/maps/api/js?key=...&callback=initializareHarta"></script>
<script type="text/javascript">
function initializareHarta()
        {
        spatiuRezervat=document.getElementById("harta")
        configurari={center:{lat:46.773579,lng:23.620014},zoom:8}
        harta=new google.maps.Map(spatiuRezervat,configurari)
        }

function marcheaza()
        {
        configurariGeocoder={address:"Club Bamboo, Strada Tuzla, Bucharest, Romania"}
        geo=new google.maps.Geocoder()
        geo.geocode(configurariGeocoder, repozitionare)
        }

function repozitionare(geoRaspuns, status)
        {
        if (status=="OK")
                {
                locatie=geoRaspuns[0].geometry.location
                configurariMarker={map:harta,position:locatie}
                semn=new google.maps.Marker(configurariMarker)
                caseta=new google.maps.InfoWindow({content:"Acesta e clubul Bamboo"})
                semn.addListener("click", function(){caseta.open(harta,semn)})
                harta.setCenter(locatie)
                harta.setZoom(15)
                }
        else {alert(status)}
        }

</script>
</head>

<body>
        <div style="border:2px solid red;width:500px;height:500px" id="harta"></div>
        <input type="button" onclick="marcheaza()" value="Apăsați aici pentru poziționare"/>
</body>
</html>
```

Observaţii:

- Nu uitați să introduceți propria cheie
- Sub hartă am creat un buton (INPUT TYPE = "button"), care va marca în mod vizual o adresă (păstrată într-un șir de caractere) și care va muta centrul hărții pe acel loc.
- Funcția marcheaza () creează un obiect Geocoder care va trimite adresa la serviciu. De asemenea, se alocă o funcție numită repozitionare (), pentru a se ocupa de răspuns la sosire:
    - creează un marcator vizual pe adresa de intrare
    - definește un eveniment clic pentru marcator: atunci când faceți clic, marcatorul va afișa o fereastră informativă cu textul "Acesta e Clubul Bamboo "
    - mișcă și mărește harta pe locația marcată
- Să ne uităm mai detaliat în funcția repozitionare ():
    - are două argumente implicite: răspunsul geocoding și status-ul
    - Orice solicitare geocoding este o cerere AJAX, prin urmare, trebuie să verificăm că am primit un răspuns corect (statusul = OK), înainte de a-l procesa. Verificați în documentație care ar putea fi alte status-uri - de exemplu, pentru a indica faptul că adresa nu a fost găsită - acest lucru se poate întâmpla dacă tastați greșit numele străzii. Ar trebui să verificați mai întâi dacă adresa poate fi găsită cu succes în site-ul obișnuit Google Maps - dacă da, atunci serviciul Geocoding o va găsi, de asemenea.
    - Răspunsul geocoding este întotdeauna un vector, deoarece este posibil ca mai multe locații să se găsească la aceeași adresă (de exemplu, dacă am uitat să includem țara și găsește aceeași stradă în mai multe orașe). Prin urmare, chiar dacă vom obține un singur răspuns avem încă nevoie să-l preluăm dintr-un vector (geoRaspuns [0]). Proprietatea geometry.location ne va da coordonatele.
    - Am creat un obiect Marker (are nevoie să cunoască harta și locația primită de la geocoder) - google.maps.Marker
    - de asemenea, am creat o casetă de informare InfoWindow cu un text inițial - google.maps.InfoWindow
    - am atribuit InfoWindow către evenimentul click al obiectului Marker (cu addListener, observați cum am inclus întreg corpul funcției ca al doilea argument în loc să creăm o nouă funcție)
    - În cele din urmă, folosim setCenter și setZoom pentru a poziționa harta pe marcator. Puteți face clic pe marcator pentru a vedea caseta de informare InfoWindow.

Acum avem toate elementele pentru a conecta exemplul cu microdate de harta Google.

6.6 Marcarea Google Maps pe baza microdatelor

```
<html>
<head>

<script src="jquery.js"></script>
<script src="jquery.microdata.js"></script>
<script async defer type="text/javascript" src="//maps.google.com/maps/api/js?key=...&callback=initializareHarta"></script>
<script type="text/javascript">

function initializareHarta()
        {
        spatiuRezervat=document.getElementById("harta")
        configurari={center:{lat:46.773579,lng:23.620014},zoom:8}
        harta=new google.maps.Map(spatiuRezervat,configurari)
        marcheazaHarta()
        }

function marcheazaHarta()
        {
        $.get("microdatasource.html",procesarePagina)
```

```
	}

function procesarePagina(raspuns)
	{
	arboreDOM=$("<div/>").append(raspuns)
	evenimente=arboreDOM.items("http://schema.org/MusicEvent")
	adrese=evenimente.map(construiesteAdrese)
	imagini=evenimente.map(culegeImagini)
	creeazaMarcaje()
	}

function construiesteAdrese()
	{
	numeClub=$(this).property("location").property("name").value()
	obiectAdresa=$(this).property("location").property("address")
	stradaClub=obiectAdresa.microdata("streetAddress")
	localitateClub=obiectAdresa.microdata("addressLocality")
	taraClub=obiectAdresa.microdata("addressCountry")
	adresaClub=numeClub+", "+stradaClub+", "+localitateClub+", "+taraClub
	return adresaClub
	}

function culegeImagini()
	{
	return $(this).property("location").property("image").value()
	}

function creeazaMarcaje()
	{
	geolocatii=[]
	casete=[] //stocăm aici toate casetele infowindows pregătite pentru markere
	semne=[] //stocăm aici toate markerele create pentru a le putea accesa oricând
	mesaje=[]
	for (i=0; i<adrese.length; i++)
		{
		mesaje[i]="<h1>Va asteptam cu drag</h1><img src='"+imagini[i]+"' height='50' width='50'/><br/>Adresa este:"+adrese[i]
		adresaGeocoder={address:adrese[i]}
		geo=new google.maps.Geocoder()
		geo.geocode(adresaGeocoder,proceseazaGeoraspuns(i))
		}
	}

function proceseazaGeoraspuns(indice)
	{
	function creeazaMarcaj(geoRaspuns,status)
		{
		if (status=="OK")
			{
			geolocatii[indice]=geoRaspuns[0].geometry.location
			configurariMarker={map:harta,position:geolocatii[indice]}
			semne[indice]=new google.maps.Marker(configurariMarker)
			casete[indice]=new google.maps.InfoWindow({content:"text initial"})
			semne[indice].addListener("click", function(){casete[indice].open(harta,semne[indice])})
			}
		else {alert(status)}
		}
	return creeazaMarcaj
	}

function arataMarcaj(i)
	{
	harta.setCenter(geolocatii[i])
	harta.setZoom(16)
	casete[i].setContent(mesaje[i])
	}

</script>

</head>
<body>
```

```
		<div style="border:2px solid red;width:500px;height:500px" id="harta"></div>
		<input type="button" onclick="arataMarcaj(0)" value="Apăsați aici pentru a localiza concertul Madona"/>
		<input type="button" onclick="arataMarcaj(1)" value="Apăsați aici pentru a localiza concertul Metallica"/>
</body>
</html>
```

Nu uitați cheia!
Acum avem 2 butoane sub hartă - fiecare dintre ele va arăta unul dintre cele două marcaje. Atunci când faceți clic pe un marcator, veți vedea un InfoWindow cu un fragment bogat de HTML (din vectorul mesaje), care arată imaginea și adresa (ambele extrase din microdate).

Observați o decizie importantă în proiectare: Acum butoanele nu creează marcatori, doar mișcă harta asupra acestora. Crearea efectivă a marcatorilor și casetelor de informare infowindows se face înainte, imediat după ce s-a creat harta (in interiorul funcției marcheazaHarta, după ce microdatele sunt colectate, în creeazaMarcaje).

Acest lucru asigură că serviciul Geocoder este contactat doar o singură dată la fiecare adresă (când harta este inițializată) și nu de fiecare dată când apăsați butoanele (la fel ca în exemplul anterior). Astfel, vom consuma mai puțin din cererile noastre AJAX gratuite către Geocoder.

De asemenea, observați că vom folosi mai mulți vectori pentru a stoca diverse obiecte:

- adrese – pentru toate adresele colectate de la microdate (exemplul nostru are doar două)
- imagini – pentru toate căile de imagine colectate de la microdate
- geolocatii - pentru toate locațiile returnate de geocoder (toate adresele)
- semne - punem aici toți markerii pe care îi creăm, pentru a-i putea accesa în orice moment
- casete – punem aici toate casetele infowindows definite pentru markeri, pentru a le putea modifica în orice moment
- mesaje- punem aici mesajele pe care dorim să le afișăm în fiecare infowindow; acestea vor fi mici fragmente de cod HTML care prezintă imaginea și adresa extrasă din microdate.

De asemenea, observați unele schimbări în modul în care este procesat răspunsul geocoder:

```
geo.geocode(adresaGeocoder,proceseazaGeoraspuns(i))
...
function proceseazaGeoraspuns(indice)
		{
		function creeazaMarcaj(geoRaspuns,status)
				{
				//...folosește geoRaspuns, status, dar și indice...
				}
		return creeazaMarcaj
		}
```

Argumentul i (care devine indice în cadrul funcției) este necesar pentru a parcurge ciclic vectorul de adrese și să apeleze separat geocoder-ul pentru fiecare adresă (nu este posibil să se trimită adrese multiple în cadrul unei cereri geocoder[54]).
Să se compare exemplul cu cel alterior, mai simplu, când apelam geocoder doar o dată:

```
geo.geocode(configurariGeocoder, repozitionare)
...
function repozitionare(geoRaspuns, status)
		{
		//...folosește doar geoRaspuns și status
		}
```

[54] Motivul pentru care geocoder-ul returnează un vector de răspunsuri nu se datorează faptului că se pot trimite mai multe adrese ci datorită faptului că pentru o anumită adresă se pot găsi mai multe locații, dacă adresa nu a fost suficient de precisă

În versiunea mai simplă, funcția care prelucrează răspunsul geocoder (repozitionare) folosește doar parametrii impliciți – răspunsul și status-ul. Este inclusă în funcția geocode ca un al doilea argument doar cu numele său și fără paranteze/argumente. Acest lucru se datorează:

- Ordinea și înțelesul argumentelor funcției sunt predefinite (răspuns, stare); nu pot fi schimbate și vor primi valorile din cererea geocoder, nu din partea programatorului;
- Includerea funcției ca argument nu este un APEL de funcție – nu o vom executa, doar o “rezervăm” (cu alte cuvinte, declarăm că VA FI executată CÂND primim răspunsul de la cererea AJAX); acest tip de funcție este numită “callback”.

În versiunea mai nouă, funcția callback (creeazaMarcaj) solicită un parametru suplimentar (indice, și anume index-ul curent în cadrul vectorului de adrese), dar NU poate fi definită cu un parametru adițional deoarece lista cu parametrii acesteia este fixă (permite doar doi parametri predefiniți care vor fi primiți în răspunsul geocoder). Pentru a face acest lucru posibil, este nevoie să încapsulăm funcția callback ca funcție intermediară, numită “closure” (proceseazăRăspuns). Această funcție intermediară va face disponibil argumentul suplimentar i / index-ul către corpul funcției creeazaMarcaj.

- Funcția “closure” proceseazăRăspuns trebuie să înlocuiască funcția “callback” creeazaMarcaj, al doilea argument în geo.geocode(), și trebuie să specifice și parametrul adițional
- Funcția “callback” creeazaMarcaj poate folosi noul parametru fără să schimbe propria listă cu parametri
- Funcția “closure” trebuie să returneze corpul funcției “callback”(fără argumente)

Această tehnică poate fi folosită în JavaScript de fiecare dată când dorim să “modificăm” o funcție a cărei semnături (lista cu parametri) este fixă – acesta este cazul tipic al funcțiilor “callback” deoarece parametrii acestora nu pot fi definiți de către programator, fiind predefiniți și trebuie să fie completați de către un anume eveniment (în cazul de față, primirea răspunsului AJAX).

6.7 Descrierea grafurilor cu JSON-LD

Creați o nouă versiune a fișierului microdatasource.html, numit jsonldsource.html. În această versiune, graful de cunoștințe ar trebui să fie stocat în format JSON-LD, care are avantajul de a fi complet separat de codul HTML, dar dezavantajul că dublează informațiile care există deja în codul HTML vizibil în browser (de exemplu adresele). Noua versiune arată astfel:

```
<html>
<head>
<script type="application/ld+json" id="sectiunejsonld">
{
"@context":{"schema":"http://schema.org/"},
"@graph":
        [
                {
                "@id":"_:event1",
                "@type":"schema:MusicEvent",
                "schema:url":{"@id":"http://themadonnaeventpage.com"},
                "schema:name":"Madonna Concert",
                "schema:performer":{"@type":"schema:MusicGroup","schema:name":"Madonna"},
                "schema:location":{
                                        "@type":"schema:Place",
                                        "schema:name":"Bamboo Club",
                                        "schema:image":{"@id":"http://localhost/Imagine1.png"},
                                        "schema:address":
                                                {
                                                "@type":"schema:PostalAddress",
                                                "schema:streetAddress":"Strada Tuzla 50",
                                                "schema:addressLocality":"Bucharest",
                                                "schema:addressCountry":"Romania"
                                                }
                                        },
                "schema:startDate":{"@type":"xsd:dateTime","@value": "2017-11-02T21:00:00+02:00"},
```

```
                    "schema:endDate":{"@type":"xsd:dateTime","@value": "2017-11-02T23:00:00+02:00"}
                    },
                    {
                    "@id":"_:event2",
                    "@type":"schema:MusicEvent",
                    "schema:url":{"@id":"http://themetallicaeventpage.com"},
                    "schema:name":"Metallica Concert",
                    "schema:performer":{"@type":"schema:MusicGroup","schema:name":"Metallica"},
                    "schema:location":{
                                        "@type":"schema:Place",
                                        "schema:name":"NOA Club",
                                        "schema:image":{"@id":"http://localhost/Imagine2.png"},
                                        "schema:address":
                                                {
                                                "@type":"schema:PostalAddress",
                                                "schema:streetAddress":"Strada Republicii 108",
                                                "schema:addressLocality":"Cluj-Napoca",
                                                "schema:addressCountry":"Romania"
                                                }
                                        },
                    "schema:startDate":{"@type":"xsd:dateTime","@value": "2017-10-13T19:00:00+02:00"},
                    "schema:endDate":{"@type":"xsd:dateTime","@value": "2017-10-13T23:00:00+02:00"}
                    }
          ]
}
</script>
</head>

<body>
  <h1>Future Events</h1>

  <div>
                    <h2><a href="http://themadonnaeventpage.com">Madonna Concert</a></h2>
                    <p> You will have fun at the concert of Madonna </p>
                    in Club Bamboo,Strada Tuzla 50,Bucharest,Romania <br/>
                    <img src="Imagine1.png" height="100" width="100"/>
                    <div>November 2, 9:00pm - 11:00pm</div>
  </div>
  <div>
                    <h2><a href="http://themetallicaeventpage.com">Metallica Concert</a></h2>
                    <p>You will have fun at the concert of Metallica</p>
                    in NOA Club,Strada Republicii 108,Cluj-Napoca,Romania<br/>
                    <img src="Imagine2.png" height="100" width="100"/>
                    <div>October 13, 7:00pm - 11:00pm</div>
  </div>

</body>
</html>
```

Observați că în codul HTML nu mai sunt microdate - unicul său scop este de a afișa unele informații în browser. In schimb, toate cunoștințele sunt descrise acum într-un graf JSON-LD, stocate în interiorul unei secțiuni SCRIPT (JSON-LD este cod valid JSON, care este un cod JavaScript valid pentru a descrie vectori asociativi!).

Vă puteți crea propriul graf JSON-LD mai ușor dacă ați converti codul anterior HTML + microdate in JSON-LD folosind distilatoare publice disponibile la[55]:
https://www.w3.org/2012/pyMicrodata/
http://rdf-translator.appspot.com/

Se recomandă totuși să se scrie propriul graf JSON-LD de câteva ori pentru familiarizare cu sintaxa. De asemenea, puteți testa dacă l-ați scris corect prin transformarea acestuia în N-quad-uri la:

[55] Dacă folosiți RDF Translator și doriți conversie în sintaxă Turtle nu o veți găsi în formatul destinație. Selectați în locul său formatul N3 (e o versiune a sintaxei Turtle cu unele reguli suplimentare).

http://json-ld.org/playground/

Codul JSON pentru partea client/consumator (consumer) este următorul:

```
<html>
<head>

<script src="jquery.js"></script>
<script async defer type="text/javascript" src="//maps.google.com/maps/api/js?key=...&callback=initializareHarta"></script>
<script type="text/javascript">

function initializareHarta()
        {
        spatiuRezervat=document.getElementById("harta")
        configurari={center:{lat:46.773579,lng:23.620014},zoom:8}
        harta=new google.maps.Map(spatiuRezervat,configurari)
        marcheazaHarta()
        }

function marcheazaHarta()
        {
        $.get("jsonldsource.html",procesarePagina)
        }

function procesarePagina(raspuns)
        {
        arboreDOM=$("<div/>").append(raspuns)
        codJSONLD=$("#sectiunejsonld",arboreDOM).html()
        cunostinte=JSON.parse(codJSONLD)
        evenimente=cunostinte["@graph"]
        adrese=$.map(evenimente,construiesteAdrese)
        imagini=$.map(evenimente,culegeImagini)
        creeazaMarcaje()
        }

function construiesteAdrese(eveniment)
        {
        numeClub=eveniment["schema:location"]["schema:name"]
        obiectAdresa=eveniment["schema:location"]["schema:address"]
        stradaClub=obiectAdresa["schema:streetAddress"]
        localitateClub=obiectAdresa["schema:addressLocality"]
        taraClub=obiectAdresa["schema:addressCountry"]
        adresaClub=numeClub+", "+stradaClub+", "+localitateClub+", "+taraClub
        return adresaClub
        }

function culegeImagini(eveniment)
        {
        return eveniment["schema:location"]["schema:image"]["@id"]
        }

function creeazaMarcaje()
        {
        geolocatii=[]
        casete=[]
        semne=[]
        mesaje=[]
        for (i=0; i<adrese.length; i++)
                {
                mesaje[i]="<h1>Va asteptam cu drag</h1><img src='"+imagini[i]+"' height='50' width='50'/><br/>Adresa este:"+adrese[i]
                adresaGeocoder={address:adrese[i]}
                geo=new google.maps.Geocoder()
                geo.geocode(adresaGeocoder,proceseazaGeoraspuns(i))
                }
        }

function proceseazaGeoraspuns(indice)
        {
        function creeazaMarcaj(geoRaspuns,status)
```

```
			{
			if (status=="OK")
				{
				geolocatii[indice]=geoRaspuns[0].geometry.location
				configurariMarker={map:harta,position:geolocatii[indice]}
				semne[indice]=new google.maps.Marker(configurariMarker)
				casete[indice]=new google.maps.InfoWindow({content:"text initial"})
				semne[indice].addListener("click", function(){casete[indice].open(harta,semne[indice])})
				}
			else {alert("ss"+status)}
			}
		return creeazaMarcaj
		}

function arataMarcaj(i)
		{
		harta.setCenter(geolocatii[i])
		harta.setZoom(16)
		casete[i].setContent(mesaje[i])
		}

</script>

</head>
<body>
		<div style="border:2px solid red;width:500px;height:500px" id="harta"></div>
		<input type="button" onclick="arataMarcaj(0)" value="Apăsați aici pentru locația Concertului Madonnei"/>
		<input type="button" onclick="arataMarcaj(1)" value="Apăsați aici pentru locația Concertului Metallica"/>
</body>
</html>
```

Observați următoarele modificări:

- Nu avem nevoie de SCRIPT care importă distilatorul microdata, dar încă avem nevoie de SCRIPT care importă JQuery (îl folosim pentru cereri AJAX);
- Conținutul grafului este extras și convertit într-un obiect JavaScript cu

```
arboreDOM = $ ( "<div />"). append (raspuns) // vom obține codul HTML complet din fișierul sursă
codJSONLD= $ ("# sectiunejsonld", arboreDOM) .html () // căutăm secțiunea SCRIPT cu ID-ul = sectiunejsonld și preluăm conținutul său
cunostinte=JSON.parse cunostinte (codJSONLD) // aici putem converti conținutul din șir de caractere în obiect JavaScript
evenimente=cunostinte["@ graph"] // aici vom avea acces la secțiunea @graph a codului JSON-LD, care ne dă întregul graf de cunoștințe
```

- Utilizarea funcției map este ușor diferită. În vechea versiune, fiecare element al vectorului evenimente a fost un element DOM (un nod în arborele DOM). Acest lucru ne-a permis să apelăm funcția map prin <array> .map (<function>) și de a folosi $ (this) în interiorul <function> pentru a ne referi la elementul DOM curent. Acum, elementele vectorului evenimente nu sunt elemente DOM, ele sunt doar simple obiecte JavaScript. În acest caz, sintaxa funcției map() este $.map (<array >, < function >) și < function > va avea un parametru predefinit obligatoriu care reprezintă elementul curent al șirului:

```
adrese=$.map(evenimente,construiesteAdrese)
imagini=$.map(evenimente,culegeImagini)
```

- De asemenea, avem o sintaxă diferită pentru modul în care sunt extrase proprietățile. În loc să folosim înlănțuirea proprietăților distilatorului microdata, am navigat prin obiectele JavaScript în funcție de etichetele găsite în structura JSON. Observați de asemenea, parametrul obligatoriu al acestor funcții (eveniment), care este necesar în loc de $ (this), pentru că acum nu procesăm noduri DOM, ci vectori JavaScript:

```
function construiesteAdrese(eveniment)
		{
		numeClub=eveniment["schema:location"]["schema:name"]
		obiectAdresa=eveniment["schema:location"]["schema:address"]
		stradaClub=obiectAdresa["schema:streetAddress"]
		localitateClub=obiectAdresa["schema:addressLocality"]
		taraClub=obiectAdresa["schema:addressCountry"]
		adresaClub=numeClub+", "+stradaClub+", "+localitateClub+", "+taraClub
```

```
            return adresaClub
            }

function culegeImagini(eveniment)
            {
            return eveniment["schema:location"]["schema:image"]["@id"]
            }
```

Observație importantă: Exemplele prezentate aici, cu două fișiere HTML citindu-și conținutul reciproc printr-o cerere AJAX, vor funcționa numai în cazul în care ambele fișiere sunt stocate în același domeniu (aici localhost). În cazul în care acestea sunt pe site-uri web diferite, consultaţi tehnicile "AJAX Cross-domain" prezentate anterior.

6.8 Interogarea Google Knowledge Graph

Am văzut cum putem utiliza mai multe servicii Google (Maps și Geocoder). În continuare, vom folosi un alt serviciu public furnizat de Google pentru aceeași cheie API. Asigurați-vă că, în panoul de control al Google API Console (https://console.developers.google.com) aveți Knowledge Graph Search API (Căutați-l și activați-l în secțiunea Library, dacă nu îl aveți).

Puteți consulta date suplimentare despre interogarea Knowledge Graph aici:
https://developers.google.com/knowledge-graph/

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script>
function initializare()
            {
            adresa="https://kgsearch.googleapis.com/v1/entities:search"
            configurari={'query':'Madonna','key':'...'}
            $.getJSON(adresa, configurari, procesareRaspuns)
            }
function procesareRaspuns(raspuns)
            {
            $.each(raspuns.itemListElement, proceseazaElement)
            }
function proceseazaElement(indice,element)
            {
            continutDeAfisat="<div>"+element["result"]["name"]+":"+element["result"]["description"]+"</div>"
            $(continutDeAfisat).appendTo(document.body)
            }
</script>
</head>
<body>
<input type="button" onclick="initializare()" value="Apăsați pentru a obține informații despre Madonna"/>
</body>
</html>
```

Nu uitați să includeți cheia.

Folosim JQuery pentru a trimite o solicitare HTTP la serviciul Knowledge Graph (de data aceasta JQuery este importat de la Google). În acest exemplu, vom face o căutare pentru Madonna și vom obține o listă de rezultate despre Madonna cântăreața, pictura și altele. Pentru a înțelege ce informații vom primi de la serviciu, trebuie să studiem structura JSON-LD. Puteți verifica acest lucru în două moduri:

- În browser (în Chrome, după ce ați executat o cerere, se va vizualiza Developers Tools – Network – XHR – Response)
- Testați manual cererea prin intermediul formularului public oferit de Google (utilizați caseta "query"): https://developers.google.com/apis-explorer/#p/kgsearch/v1/kgsearch.entities.search

```
{
 "@context": {
  "@vocab": "http://schema.org/",
  "goog": "http://schema.googleapis.com/",
  "EntitySearchResult": "goog:EntitySearchResult",
  "detailedDescription": "goog:detailedDescription",
  "resultScore": "goog:resultScore",
  "kg": "http://g.co/kg"
 },
 "@type": "ItemList",
 "itemListElement": [
  {
   "@type": "EntitySearchResult",
   "result": {
    "@id": "kg:/m/01vs_v8",
    "name": "Madonna",
    "@type": [
     "Thing",
     "Person"
    ],
    "description": "Singer-songwriter",
    "image": {
     "contentUrl": "http://t0.gstatic.com/images?q=tbn:ANd9GcTtSYFxXeE6Q092a0w8fbsLwztopeXRL1n5uudTbxalHy8sSk4N",
     "url": "https://en.wikipedia.org/wiki/Madonna_(entertainer)",
     "license": "http://creativecommons.org/licenses/by-sa/3.0/"
    },
    "detailedDescription": {
     "articleBody": "Madonna Louise Ciccone is an American singer, songwriter, dancer, actress, and businesswoman. She achieved popularity by pushing the boundaries of lyrical content in mainstream popular music and imagery in her music videos, which became a fixture on MTV. ",
     "url": "https://en.wikipedia.org/wiki/Madonna_(entertainer)",
     "license": "https://en.wikipedia.org/wiki/Wikipedia:Text_of_Creative_Commons_Attribution-ShareAlike_3.0_Unported_License"
    },
    "url": "http://www.madonna.com/"
   },
   "resultScore": 109.096672
  },
  {
   "@type": "EntitySearchResult",
   "result": {.............................................},
........................................
]
}
```


Observați următoarele:

- rezultatele trebuie să fie extrase dintr-un vector numit itemListElements
- URI-urile sunt formate prin adăugarea prefixul kg: la Freebase ID-ul de la wikidata (https://www.wikidata.org/wiki)
- în exemplul nostru am afișat numai numele și descrierea proprietăților

O altă modalitate de a interoga cunoștințele de la Google este de a accesa direct baza de cunoștințe Wikidata (Wikidata este similar cu DBPedia și este nucleul Knowledge Graph Google). Puteți consulta diverse metode pentru a obține grafuri de cunoștințe din Wikidata la:
https://www.wikidata.org/wiki/Wikidata:Data_access

De exemplu, la adresa următoare este un serviciu SPARQL public: https://query.wikidata.org/

Puteți obține, de asemenea, RDF brut (fișiere uriașe pe care să le încărcați în propria bază de cunoștințe dacă aveți suficientă memorie RAM la dispoziție):
http://tools.wmflabs.org/wikidata-exports/rdf/

6.9 Distilarea grafurilor RDF în Python

Următorul fragment conține cod HTML distilabil folosind tehnica microdatelor (microdataexample.html).

```
<div itemscope="itemscope" itemid="http://expl.at#Avatar" itemtype ="http://schema.org/Movie" >
<h1 itemprop="name">Avatar</h1>
Directed by:
        <span itemprop="director" itemid="http://expl.at#JamesCameron"
                itemscope="itemscope" itemtype="http://schema.org/Person" >
                <span itemprop="name">James Cameron</span>
                <meta itemprop="birthDate" content="August 16, 1954"/>
                 <link itemprop="url" href="https://en.wikipedia.org/wiki/James_Cameron"/>
                <span itemprop="spouse" itemscope="itemscope" itemid="http://expl.at#SuzyAmis"
                        itemtype="http://schema.org/Person"/>
        </span>
</div>
```

În browser veți vedea următoarele

Avatar

Directed by: James Cameron

Figura 40 Vizualizarea în browser a paginii microdataexample.html

Testați același exemplu cu un distilator microdata:

https://www.w3.org/2012/pyMicrodata/#distill_by_input
(pe parcursul anului 2017 nu a funcționat, fiind în curs de reimplementare; adăugăm dedesubt și alte variante)

http://rdf-translator.appspot.com/ (aici nu apare formatul Turtle în lista Output, dar apare formatul N3 care e similar)
http://rdf.greggkellogg.net/distiller

În urma distilării ar trebui să obțineți:

```
@prefix schema: <http://schema.org/> .

<http://expl.at#Avatar> a schema:Movie;
   schema:director <http://expl.at#JamesCameron>;
   schema:name "Avatar" .

<http://expl.at#JamesCameron> a schema:Person;
   schema:birthDate "August 16, 1954";
   schema:name "James Cameron";
   schema:spouse <http://expl.at#SuzyAmis>;
   schema:url <https://en.wikipedia.org/wiki/James_Cameron> .

<http://expl.at#SuzyAmis> a schema:Person .
```

Pentru a permite accesarea acestei pagini prin HTTP, puneți-o în XAMPP/Apache (htdocs) într-un fișier cu numele microdataexample.html. Ulterior vom distila această pagină în Python, accesând-o prin această adresă. Testați accesarea paginii în browser cu http://localhost/microdataexample.html

În continuare vom exemplifica tehnica RDFa care e o versiune mai sofisticată a tehnicii microdata, folosindu-se de alte atribute HTML.

Să se scrie următorul cod într-o pagină numită rdfapage.html pe care o veți salva în serverul Apache (pentru a deveni accesibilă la adresa http://localhost/rdfapage.html).

```
<html>

<body prefix="x: http://expl.at#">
```

```
<table border="1">

<tr>
<td style="color:red">Name</td></tr>
<tr>
<td about="x:Andrew">
<span property="foaf:name">Andrew Smith</span>
<span rel="foaf:knows">
		<span resource="x:Mary"></span>
		<span resource="x:Anna"></span>
		<span resource="x:George"></span>
</span>
</td>
</tr>

<tr>
<td about="x:Mary">
<span property="foaf:name">Mary Smith</span>
<span rel="foaf:knows">
		<span resource="x:Andrew"></span>
		<span resource="x:Anna"></span>
</span>
</td>
</tr>

<tr>
<td about="x:Anna">
<span property="foaf:name">Anna Smith</span>
<span rel="foaf:knows">
		<span resource="x:Andrew"></span>
		<span resource="x:Mary"></span>
</span>
</td>
</tr>

<tr>
<td about="x:George">
<span property="foaf:name">George Smith</span>
<span rel="foaf:knows">
		<span resource="x:Andrew"></span>
</span>
</td>
</tr>

</table>
</body>
</html>
```

Observaţi atributele RDFa prin care fiecare celulă HTML a fost adnotată. Precum în tehnica microdata, aceste atribute pot fi ataşate oricărui element HTML (DIV, SPAN etc.) pentru a indica despre ce e vorba în elementul respectiv, cu ajutorul unor proprietăţi (alese dintr-o terminologie cunoscută – schema.org, FOAF, etc.). Atributele RDFa folosite în acest exemplu sunt:

- ABOUT indică identificatorul URI al entităţii menţionate în celulă (deci subiectul afirmaţiilor RDF);
- SPAN PROPERTY indică proprietatea, în cazul în care aceasta e un atribut al cărei valoare e text vizibil în browser (aici e vorba de numele afişat în celula tabelului, declarat ca valoare a proprietăţii foaf:name);
- SPAN REL indică tot o proprietate, în cazul în care aceasta exprimă o relaţie cu alt URI (aici relaţiile foaf:knows); cu SPAN RESOURCE indicăm care este acel URI (obiectul afirmaţiilor). Aceste elemente SPAN nu conţin nimic, deci nu vor influenţa ceea ce se vede în browser – rolul lor e doar să ofere relaţii interogabile clienţilor sau motoarelor de căutare care vor distila acest cod.

Observați că am definit doar prefixul x:, deși în cod se folosește și prefixul foaf: (pentru foaf:name, foaf:knows două proprietăți din terminologia FOAF – „friend of a friend” – folosită la descrieri de persoane

şi relaţii sociale). Omiterea definirii prefixului foaf: e posibilă deoarece W3C a stabilit o listă cu anumite prefixe ce nu mai trebuie declarate (cel puţin în RDFa). Lista este disponibilă la adresa:

http://www.w3.org/2011/rdfa-context/rdfa-1.1

Toate distilatoarele RDFa trebuie să recunoască automat aceste prefixe şi adresele de domeniu asociate lor.

Un utilizator uman care deschide pagina în browser ar trebui să vadă un tabel HTML cu o coloană de nume:

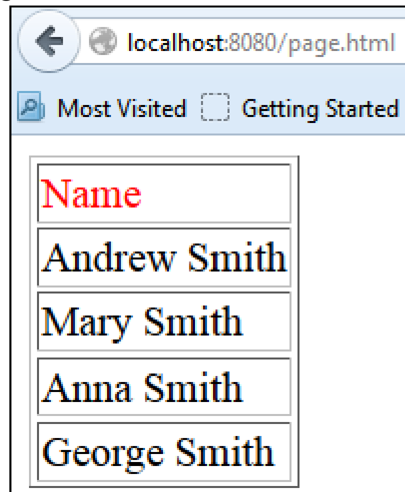


Figura 41 Vizualizarea în browser a paginii page.html

În schimb un software cu capacităţi de distilare (motor de căutare, plug-in de browser, aplicaţie ce „vizitează” pagina prin cereri HTTP) nu va vedea doar o listă de stringuri, ci o reţea socială care exprimă faptul că unii din aceşti indivizi se cunosc între ei. Acest lucru se poate afla prin **distilare**.

Exemple de instrumente on-line unde puteți testa distilarea RDFa (copiați codul integral în caseta din stânga):

http://rdf-translator.appspot.com/ (aici în loc de Turtle apare formatul N3, care e similar)
https://rdfa.info/play/
http://rdf.greggkellogg.net/distiller
https://www.w3.org/2012/pyRdfa/#distill_by_input

Încărcați fișierul HTML la unul din aceste servicii, alegeți Turtle (sau N3) ca sintaxă de ieşire și veți primi un fișier precum cel de mai jos:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix x: <http://expl.at#> .

x:George foaf:knows x:Andrew; foaf:name "George Smith" .
x:Anna foaf:knows x:Andrew, x:Mary; foaf:name "Anna Smith" .
x:Mary foaf:knows x:Andrew, x:Anna; foaf:name "Mary Smith" .
x:Andrew foaf:knows x:Anna, x:George, x:Mary; foaf:name "Andrew Smith" .
```

Prin distilare acest graf va deveni accesibil pentru procesare oricărui client capabil să proceseze grafuri RDF. În continuare vom folosi linia de comandă Python pentru a realiza astfel de distilări. Principala librărie care se ocupă cu procesare de grafuri în Python este rdflib, având un rol similar cu al librăriei EasyRDF exemplificată deja în PHP. Şi acea librărie funcţionează ca distilator pentru PHP, dacă se indică rdfa ca sintaxă de intrare:

```
$graf=new EasyRdf_Graph();
$graf->load("http://localhost/rdfapage.html","rdfa");
```

În continuare vom executa distilarea exemplului cu microdate folosind linia de comandă Python pe post de client HTTP și distilatorul oferit de librăria rdflib[56] (care necesită şi instalarea librăriei html5lib[57] pentru partea de distilare).

Folosim pentru exemplificare Python 3.6, versiune a cărei instalare vine cu manageri de librării externe gata instalați (pip și easy_install). Versiuni mai vechi pot necesita instalarea separată a unui manager de pachete, acesta fiind inclus în kit-ul de instalare Python abia în versiuni recente[58].

Python 3.6 se poate procura de la adresa
https://www.python.org/downloads/

În timpul instalării bifați și opțiunea "Add Python to PATH".
Pentru a instala librăriile necesare următoarelor exerciții, folosiți în linia de comandă Windows managerul pip (oriunde în Windows, dacă Python s-a adăugat la variabila de mediu PATH):

```
pip install rdflib
```

apoi

```
pip install html5lib
```

Intrați în linia de comandă IDLE oferită de Python, unde presupunem că aveți librăriile instalate și sunt gata de import. Realizăm distilarea fișierului anterior:

```
>>> import rdflib
>>> g=rdflib.Graph()
>>> g.parse(location="http://localhost/microdataexample.html")
>>> contents=g.serialize(format="turtle")
>>> print(contents)
```

Observații:

- În unele versiuni ale librăriei rdflib, în funcţia parse() ar putea fi necesară şi indicarea sintaxei din care se face distilarea, în maniera:
  ```
  g.parse(location="http://localhost/microdataexample.html", format="microdata")
  ```
- Rezultatul distilării a fost convertit în format turtle şi stocat într-un şir de caractere de tip byte (înaintea primului apostrof e prefixat cu litera b). Dacă dorim o afişare mai prietenoasă, care să permită caractere Unicode şi în care codurile \n să fie convertite în salturi la rând nou, mai trebuie folosită şi funcţia decode:

```
>>> print(contents.decode())
```

Atenţie, aceasta e o particularitate Python 3 care, spre deosebire de Python 2, oferă două tipuri de şiruri de caractere: *stringurile de tip byte* (semnalate prin litera b înaintea apostrofului şi conţinând doar caractere ASCII) şi *stringurile unicode* (cu caractere universale). Conversia între stringuri byte şi stringuri unicode are loc cu decode() (şi în sens invers cu encode()), o operaţie la care trebuie să fie atenţi cei care trec de la Python 2 la Python 3.

Observați în rezultatul distilării câteva afirmații care nu erau prezente în codul HTML, dar sunt generate de distilator. E vorba de proprietatea item, a cărei valoare e lista subiectelor detectate, şi proprietatea rdfa:usesVocabulary a cărei valoare indică terminologia detectată în codul microdata.

```
<http://localhost/microdataexample.html>    <http://www.w3.org/ns/md#item> (:Avatar);
                                            <http://www.w3.org/ns/rdfa#usesVocabulary> <http://schema.org/>.
```

[56] https://rdflib.readthedocs.io/en/stable/
[57] https://html5lib.readthedocs.io/en/latest/
[58] Detalii pentru instalarea manuală a unui manager de pachete aici: https://packaging.python.org/installing/ (după cum se arată acolo, instalarea manuală nu mai e necesară în Python 2 începând cu v2.7.9, iar în Python 3 începând cu v3.4). O discuție mai detaliată despre diferențele dintre diverși manageri de pachete disponibili găsiți la http://xahlee.info/python/python_whats_pip_easyinstall_setuptools.html (easy_install e "clasic", dar pip e mai popular în Python 3).

Serializarea ne oferă graful distilat sub formă de şiruri de caractere, ceea ce e util în a confirma dacă am obţinut graful dorit. În continuare convertim graful într-un array de afirmaţii, folosind funcţia triples cu şablonul (None,None,None), echivalent cu o interogare SPARQL pe şablonul {?x ?y ?z}:

```
>>> contents2=[x for x in g.triples((None,None,None))]
>>> for (x,y,z) in contents2:
        print(x,y,z,sep=",")
```

Am afişat conţinutul grafului cu termenii separaţi prin virgule. Atenţie, în Python 3 print() este o funcţie şi nu o comandă (ca în Python 2), oferind prin diverse argumente (precum sep) posibilitatea de a defini caractere de delimitare între valorile afişate.

Aceeaşi clasă Graph ne oferă şi un distilator RDFa, pe care îl vom aplica pe al doilea fişier creat la începutul acestui capitol:

```
>>> import rdflib
>>> g=rdflib.Graph()
>>> g.parse(location="http://localhost/rdfapage.html",format="rdfa")
>>> contents=g.serialize(format="turtle")
>>> print(contents.decode())
```

În continuare vom folosi funcţiile rdflib pentru a extrage diverse informaţii din graful obţinut prin distilare:

```
>>> contents=[x for x in g.triples((None,None,None))]
>>> for i in contents:
        print(i)
```

Acum contents nu mai este un string obţinut prin serializare ci este o listă de tripleţi, adică un vector cu toate afirmaţiile distilate. În fiecare afirmaţie termenii vor apărea calificaţi cu rdflib.term. Pentru a avea o afişare mai prietenoasă putem aplica o conversie în string a fiecărui termen:

```
>>> for (s,p,o) in contents:
        print((str(s),str(p),str(o)))
```

Mai departe extragem doar numele din graful distilat, cunoscând faptul că acestea au fost declarate cu foaf:name:

```
>>> names=[x for x in g.objects(None,rdflib.URIRef("http://xmlns.com/foaf/0.1/name"))]
>>> for i in names:
        print(i)
```

Observaţi folosirea lui None ca locţiitor pentru "orice subiect". Apoi proprietatea foaf:name a fost indicată prin URI-ul său complet, calificat de rdflib.URIRef. Pentru a evita acest mod incomod de a scrie URI-uri complete, putem înregistra un prefix. În cazul FOAF (şi pentru alte terminologii populare) prefixul e deja înregistrat în rdflib şi trebuie doar importat în maniera de mai jos:

```
>>> from rdflib.namespace import FOAF
>>> names=[x for x in g.objects(None,FOAF.name)]
>>> for x in names:
        print(x)
```

Atenţie, chiar dacă pe ecran obţinem o listă de stringuri, rdflib îşi stochează toţi termenii din graf, indiferent de natura lor, ca obiecte Python (funcţia print le converteşte la afişare în stringuri). Putem verifica asta dacă afişăm direct vectorul names:

```
>>> names
[rdflib.term.Literal('George Smith'), rdflib.term.Literal('Andrew Smith'), rdflib.term.Literal('Mary Smith'), rdflib.term.Literal('Anna Smith')]
```

Observaţi obiectele de tip rdflib.term - chiar dacă din punct de vedere RDF e vorba de valori de tip string, Python le gestionează ca pe nişte obiecte. Pentru a obţine efectiv un vector de stringuri native mediului Python, trebuie să aplicăm termenilor din graf funcţia de conversie în string:

```
>>> stringnames=[str(x) for x in names]
>>> stringnames
['George Smith', 'Andrew Smith', 'Mary Smith', 'Anna Smith']
```

Acum noul vector nu e doar afişat ca stringuri, ci chiar stochează stringuri (la fel procedăm pentru a stoca URI-uri ca stringuri Python).

Observaţi diferenţa între g.triples() – care returnează o listă de afirmaţii (fiecare afirmaţie fiind un tuplu de 3 elemente) – şi g.objects() – care returnează o listă de termeni (obiecte ale afirmaţiilor). Aşadar dacă am folosi g.triples() pentru a obţine lista numelor, trebuie să ţinem cont de modul în care ni se returnează rezultatele, extrăgând numele de pe poziţia 2 a fiecărui tuplu:

```
>>> statements=[x for x in g.triples((None,FOAF.name,None))]
>>> statements
>>> stringnames=[str(x[2]) for x in statements]
>>> stringnames
['George Smith', 'Andrew Smith', 'Mary Smith', 'Anna Smith']
```

O variantă mai scurtă a funcţiei triples() este cea de mai jos, ce foloseşte o codificare aparte permisă doar în Python, în care caracterul : separă cei trei termeni ai unei afirmaţii Prin urmare în loc de:

```
g.triples((None,FOAF.name,None))
```

se poate scrie

```
g[:FOAF.name:]
```

(pe poziţia termenilor None nu se scrie nimic, iar : se foloseşte ca separator între cei trei termeni)

```
>>> pairs=[x for x in g[:FOAF.name:]]
>>> stringnames=[str(ob) for (sub,ob) in pairs]
>>> stringnames
['Mary Smith', 'Anna Smith', 'Andrew Smith', 'George Smith']
```

O diferenţă faţă de triples() e că nu se mai returnează afirmaţii complete, ci doar termenii lipsă (aici perechi cu subiectele şi obiectele conectate de foaf:name).

Observaţie: În Python există noţiunea de generator, ce poate fi considerat un vector care poate fi parcurs o singură dată, după care se distruge (din raţiuni de gestiune mai eficientă a memoriei). Aceasta poate crea surprize începătorilor care vor încerca să parcurgă un array de două ori, dar a doua oară îl vor găsi gol! În aceste exerciţii, funcţiile triples(), objects() dar şi sintaxa specială g[s:p:o] returnează generatori, de aceea imediat îi copiem într-un vector normal cu construcţii de forma [x for x in ...].

Am arătat că prefixul foaf este deja înregistrat de rdflib. Pentru termenii creaţi de noi (URI-ul lui Andrew, George etc.) trebuie să înregistrăm propriul prefix. În continuare extragem numele lui Andrew:

```
>>> x=rdflib.Namespace("http://expl.at#")
>>>name=g[x.Andrew:FOAF.name:]
>>>stringname=[str(x) for x in name]
>>>stringame
['Andrew Smith']
```

Observaţi că deşi am extras o singură valoare, ea se extrage ca vector (Andrew ar putea avea mai multe nume). Construcţia folosită aici poate conţine şi înlănţuiri de proprietăţi, ca în SPARQL. Mai jos extragem lista cu numele cunoscuţilor lui Andrew folosind pe poziţia proprietăţii înlănţuirea foaf:knows/foaf:name:

```
>>> names=g[x.Andrew:FOAF.knows/FOAF.name:]
>>> stringnames=[str(x) for x in names]
>>> stringnames
['Mary Smith', 'Anna Smith', 'George Smith']
```

Pe parcursul următoarelor exerciţii se vor evidenţia trei moduri de a aduce grafuri în Python:

- Prin funcţia parse() a clasei Graph din librăria rdflib – aceasta e metoda cea mai simplă, demonstrată deja. Aceasta trimite cereri HTTP de tip GET cu configurări implicite (şi aplică distilare dacă e nevoie), deci funcţionează similar cu funcţia load() din EasyRDF (sau orice altă funcţie ce accesează o adresă URL);
- Prin librăria SPARQLWrapper – aceasta e metoda similară cu folosirea EasyRDF_SPARQL_Client în PHP, oferind funcţii deja pregătite pentru a trimite interogări SPARQL la server;
- Prin librăria urllb pentru cereri HTTP configurate în toate detaliile necesare - aceasta e metoda similară cu folosirea funcţiilor HTTP din PHP (am văzut deja clasa EasyRDF_Http_Client)

În exerciţiile următoare vom construi pagina RDFa cu tabelul de nume în mod dinamic, preluând numele şi relaţiile dintr-o bază de grafuri RDF4J. Dar pentru asta mai întâi trebuie să ne familiarizăm cu modul de construire a site-urilor în Python. Cel mai simplu mod este să folosim frameworkul CherryPy care ne oferă propriul server şi mecanismul şabloanelor Mako (similare cu şabloanele View din Laravel sau alte frameworkuri MVC).

Creaţi un site Web simplu în Python

Site-urile Web pot fi realizate în Python folosind o diversitate de framework-uri, printre care se numără CherryPy. Vom instala librăriile CherryPy executând următoarea comandă în linia de comandă Windows:

```
pip install cherrypy
```

Creați un director C:\pysite pentru a salva toate fișierele site-ului. În acest director, vom crea un fișier de configurare cu numele config.txt cu următorul conținut:

```
[global]
server.socket_host: "localhost"
server.socket_port: 9999
```

Acest fișier definește adresa și portul unde serverul CherryPy poate fi accesat.

În directorul C:\pysite creați un prim site de test. Pentru aceasta, introduceți următorul cod într-un fișier numit test.py:

```
import cherrypy
class MySite:
    @cherrypy.expose
    def index(self):
        return "This is the default page of my test site"
    @cherrypy.expose
    def otherpage(self,data):
        return "This is another page. During access it received the value "+data
cherrypy.quickstart(MySite(),config="config.txt")
```

A se observa:

- Acest site web este creat ca o clasă Python;
- Fiecare pagină a site-ului este creată ca metodă a clasei, ce preia ca argument obligatoriu cuvântul ”self” plus parametrii GET pe care vrem să îi poată accepta pagina atunci când o accesăm prin metoda GET. Astfel de metode obiectuale ce au rolul de a genera la cerere pagini Web poartă nume de **controllere** (în filozofia MVC: model-view-controller). Aici am definit două pagini:
    - cea implicită (*index*)
    - *otherpage* (care la accesare aşteaptă o valoare prin GET, în parametrul cu numele *data*);
- Fiecare pagină e precedată de decoratorul *expose*, care asigură vizibilitatea paginii (putem avea şi pagini interne inaccesibile);
- Ultima linie va porni serverul CherryPy şi în acelaşi timp va face accesibil site-ului, folosind detaliile din fișierul de configurare.

Deschideți linia de comandă Windows și navigați în directorul site-ului C:\pysite. Executați fișierul test:

```
cd C:\pysite
C:\pysite> test.py
```

Pentru pagina *index* nu e necesară indicarea numelui său, e suficientă adresa localhost:9999:

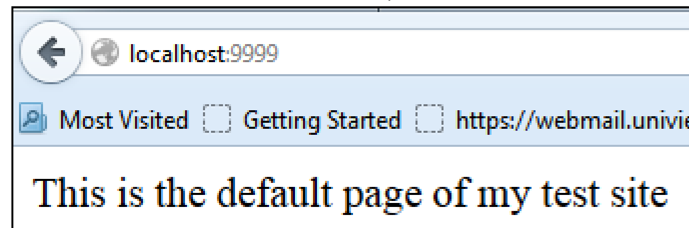


Figura 42 Pagina ”index” în browser

La accesarea paginii *otherpage* trebuie să specificăm și parametrul GET (*data*) deoarece este aşteptat de pagină. Tot ce face pagina este să concateneze acel parametru la un text:



Figura 43 Vizualizarea paginii ”otherpage” prin specificarea parametrului ”data”

Opriți serverul CherryPy prin Ctrl+C în linia de comandă.

Creaţi un site Python pe bază de şabloane Mako

În exemplul anterior, codul HTML se scrie sub formă de stringuri în acele comenzi *return*, din cadrul fiecărei pagini. O metodă mai sofisticată de tip MVC, este cea prin care codul HTML e proiectat sub forma unor şabloane (**views**, conform arhitecturii MVC) folosind librăria Mako. O veți instala prin executarea următoarei comenzi în linia de comandă Windows:

```
pip install mako
```

Creați un șablon (view) Mako pentru partea de front-end: salvați-l ca fișier myfrontend.txt tot în directorul C:\pysite) cu următorul conținut:

```
I got the following value: <br/>
<strong> ${data} </strong>
```

Observaţi că un şablon Mako e un fişier text ce conţine cod HTML şi la unele poziţii indică prin semnul $ că se vor insera în mod dinamic variabile oferite de controllerele CherryPy. Acest şablon va fi returnat în browser în locul string-ului din exemplul anterior, împreună cu datele necesare concatenate în cadrul şablonului.

Creați un site Web nou (salvați-l sub denumirea makotest.py în directorul C:\pysite) cu următorul conținut:

```
import cherrypy
import mako.template
class Site:
    @cherrypy.expose
    def index(self):
        frontend=mako.template.Template(filename="myfrontend.txt")
        return frontend.render(data="123456")
cherrypy.quickstart(Site(),config="config.txt")
```

Observaţi că de data aceasta cu *return* nu am mai construit un string, ci am apelat funcţia de procesare a şablonului Mako, indicând şi variabila pe care şablonul le aşteaptă (aceeaşi variabilă *data*, de data asta cu o valoare constantă atribuită direct în cod).

Lansați noul site web prin tastarea în linia de comandă Windows (din directorul site-ului web):
C:\pysite> makotest.py

Accesați pagina de pornire a site-ului și ar trebui să vedeţi rezultatul inserării constantei *data* în codul şablonului Mako:

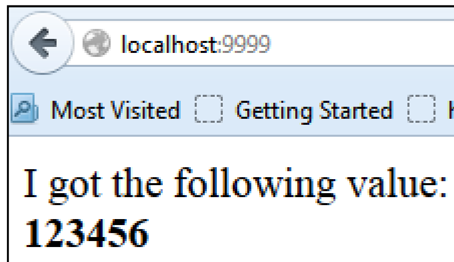


Figura 44 Afișarea valorii variabilei "data" în șablonul Mako

Opriți serverul (Ctrl+C în linia de comandă Windows)

Modificați site-ul web (creaţi un fişier makotest2.py) astfel încât să transmită către șablonul Mako mai mult de o singură valoare, sub forma unei liste Python:

```
import cherrypy
import mako.template
class Site:
   @cherrypy.expose
   def index(self):
      frontend=mako.template.Template(filename="myfrontend2.txt")
      return frontend.render(data=[1,2,3,4])
cherrypy.quickstart(Site(),config="config.txt")
```

Modificați șablonul Mako (myfrontend.txt) astfel încât să afișeze toate valorile primite într-o buclă și să verifice dacă acestea sunt numere pare sau impare, generând câte un string corespunzător celor două situaţii:

```
%for x in data:
        the received value is ${x}, which is
                %if x%2==0:
                        an even number
                %else:
                        an odd number
                %endif
                <br/>
%endfor
```

Observaţi că şabloanele Mako pot fi extinse cu elemente de programare (liniile precedate de caracterul %) pentru a asigura o procesare a datelor înainte de inserarea lor în codul HTML.

Lansați site-ul web în linia de comandă Windows (în cadrul directorului cu site-ul web):
C:\pysite> makotest2.py

Verificați în browserul web:

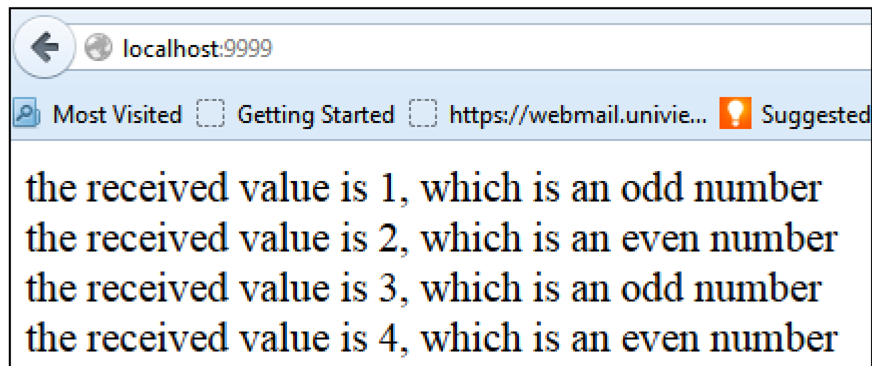


Figura 45 Afișarea mai multor valori într-un șablon Mako

Creați un site care să afișeze pagina HTML+RDFa folosită anterior. Presupunem că datele sunt deja disponibile în dicţionare Python

Presupunem că toate datele sunt disponibile în două dicționare Python:

- *namedict* conţine perechi între indivizi şi numele lor afişabile
- *relationdict* conţine pentru fiecare individ o listă cu persoanele pe care le cunoaşte.

Vom crea site-ul web în C:\pysite cu numele staticdata.py:

```
import cherrypy
import mako.template

class Site:
   @cherrypy.expose
   def index(self):
      namedict={"x:Andrew":"Andrew Smith", "x:Mary":"Mary Smith", "x:Anna":"Anna Smith", "x:George":"George Smith"}
      relationdict={"x:Andrew":["x:Anna","x:George","x:Mary"], "x:Mary":["x:Andrew","x:Anna"], "x:Anna":["x:Andrew","x:Mary"],
"x:George":["x:Andrew"]}
      address="x: http://expl.at#"
      frontend=mako.template.Template(filename="table.txt")
      return frontend.render(names=namedict,relations=relationdict,pref=address)
cherrypy.quickstart(Site(),config="config.txt")
```

Observaţi cele două dicţionare iniţializate direct în pagină împreună cu adresa de domeniu pentru a forma URI, apoi sunt transmise spre şablonul Mako ce va trebui să construiască din ele cod HTML+RDFa.

Creați un șablon Mako (”table.txt”) care parcurge într-o buclă dicționarul cu relații și generează pentru fiecare individ o celulă de tabel (cu numele) plus o serie de elemente SPAN cu relaţiile sale sociale:

```
<html>
<body prefix="${pref}">
<table border="1">
<tr><td style="color:red">Name</td></tr>
%for individual in relations:
        <tr>
        <td about="${individual}">
        <span property="foaf:name">${names[individual]}</span>
        <span rel="foaf:knows">
        %for knownperson in relations[individual]:
                <span resource="${knownperson}"/>
        %endfor
        </span>
        </td>
        </tr>
%endfor
</table>
</body>
</html>
```

Observaţi structura HTML prin care s-a extins fiecare celulă de tabel:

- cu ABOUT s-a indicat URI-ul individului din fiecare celulă (deci subiectul afirmaţiilor);
- cu SPAN PROPERTY s-a declarat numele acestora, acesta fiind totodată textul vizibil din celulă;
- cu SPAN REL s-au declarat relaţiile sociale, invizibile în browser (foaf:knows);
- cu SPAN RESOURCE s-au declarat obiectele acelor relaţii sociale (pe cine cunoaşte fiecare). Observaţi că acest SPAN nu are conţinut, deci nu va avea nici un efect vizual în browser, rolul său fiind doar de a stoca cunoştinţe în pagină, într-un mod invizibil vizitatorilor umani.

Lansați noul site web în linia de comandă Windows din directorul ce conține site-ul web (închideți site-ul anterior cu Ctrl+C):

```
C:\pysite> staticdata.py
```

Verificați site-ul în cadrul unui browser.

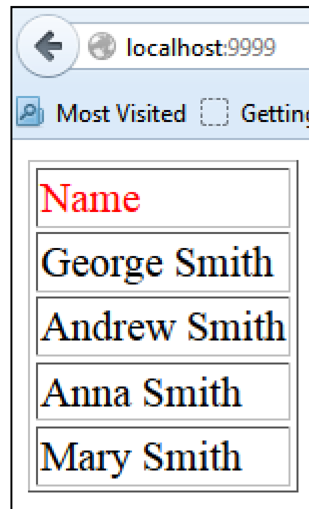


Figura 46 Vizualizare pagină în browser

Verificați și codul sursă HTML pentru a vedea dacă s-au generat corect extensiile RDFa la codul HTML. Luaţi cu copy-paste codul sursă din browser şi testaţi-l într-un distilator RDFa.

Eventual, testaţi distilarea şi din linia de comandă Python IDLE:

```
>>> import rdflib
>>> g=rdflib.Graph()
>>> g.parse("http://localhost:9999",format="rdfa")
>>> distilled=g.serialize(None,format="nt")
>>> print(distilled.decode())
```

Ar trebui să se vadă afirmațiile despre relaţiile sociale dintre indivizi, în format N-triples.

Modificați exemplul anterior pentru ca relaţiile sociale să nu fie preluate din dicţionare statice, ci să fie interogate dinamic din RDF4J

În primul rând, trebuie să facem disponibile relaţiile sociale în RDF4J, pentru a fi interogate de orice clienţi ai bazei noastre de cunoştinţe. Totuși, nu vom încărca decât următoarele afirmaţii:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix x: <http://expl.at#> .

x:Mary foaf:name "Mary Smith" .
x:Andrew foaf:knows x:Anna, x:Mary; foaf:name "Andrew Smith" .
x:George foaf:knows x:Andrew; foaf:name "George Smith" .
x:Anna foaf:knows x:Mary; foaf:name "Anna Smith" .
```

Ceea ce lipsește sunt relațiile reciproce (*x:Anna foaf:knows x:Andrew*). Am stocat doar relații unidirecționale (x:Andrew foaf:knows x:Anna) și vom genera inversele acestora în mod dinamic, prin aplicarea unor reguli. În acest mod păstrăm baza de cunoştinţe la o dimensiune minim necesară şi generăm afirmaţii noi doar când e nevoie de ele pentru front-end.

Încărcați acest exemplu în RDF4J într-o bază de cunoștințe numită *social*. În continuare vom construi în mod dinamic dicţionarele din exemplul precedent folosite ca sursă de date graful *social* din RDF4J.

Întâi verificăm funcționarea comenzilor necesare pentru a construi dicționarele *namedict* și *relationdict* din exemplul anterior. O facem pas cu pas, în linia de comandă Python IDLE.

Începem prin a realiza importurile necesare (dacă nu aţi închis consola IDLE nu le mai repetaţi, însă în momentul în care le includem în site ele vor trebui să apară oricum):

```
>>> import rdflib
>>> from rdflib.namespace import FOAF
```

Aducem într-un graf rdflib tot conţinutul bazei de grafuri *social* din RDF4J, printr-o cerere GET la adresa ce returnează conţinutul integral al bazei de date. Observaţi că folosim aceeaşi funcţie parse() pe care am folosit-o şi la distilare (în realitate funcţia parse accesează o adresă URL prin metoda GET şi extrage graful pe care îl găseşte acolo, indiferent că e obţinut prin distilare sau e disponibil direct într-o sintaxă RDF):

```
>>> g=rdflib.Graph()
>>> g.parse("http://localhost:8080/rdf4j-server/repositories/social/statements")
```

Construim dicţionarul namedict din exemplul precedent:

```
>>> namemappings=g[:FOAF.name:]
>>> namedict={g.qname(x):str(y) for (x,y) in namemappings}
>>> namedict
{'x:Mary': 'Mary Smith', 'x:George': 'George Smith', 'x:Anna': 'Anna Smith', 'x:Andrew': 'Andrew Smith'}
```

Observaţii:

- am extras din graf subiectele şi obiectele din afirmaţiile cu predicatul foaf:name (deci am extras indivizii şi numele acestora);
- deoarece indivizii vor fi URI, îi trecem prin funcţia qname() care returnează ca string forma prefixată a unui URI ("x:Mary" în loc de "http://expl.at#Mary"); numele le trecem prin funcţia str() pentru a obţine forma simplă a numelor, ca stringuri (am arătat că în mod implicit Python stochează nodurile unui graf ca obiecte, chiar şi dacă e vorba de noduri care nu sun URI ci valori string);
- am folosit o formă concisă de ciclu FOR pentru a construi dicţionarul final.

Mai departe trebuie construit dicţionarul relationdict din exemplul precedent:

```
>>> myquery="construct {?a foaf:knows ?b.?b foaf:knows ?a} where {?a foaf:knows ?b}"
>>> result=g.query(myquery)
```

Observaţii:

- Am construit o interogare SPARQL care să construiască relaţiile foaf:knows în ambele direcţii, deoarece asta e forma în care vrem să le prezentăm în codul distilabil. Amintiţi-vă că în RDF4J le-am stocat într-o singură direcţie pentru a minimiza dimensiunea bazei de date. Aici practic avem de a face cu o regulă care generează relaţii foaf:knows inverse în vederea inserării lor în codul distilabil, fără a le şi salva în baza de date!
- Reţineţi că în mod normal interogările trebuie să includă şi declararea prefixelor folosite în interogare, dar pentru prefixe populare precum foaf: librăria rdflib ne scuteşte de acest efort.

Amintiţi-vă că în mod normal interogările CONSTRUCT ar trebui să returneze un graf care să fie mai departe procesat cu metode specifice clasei Graph(). Am demonstrat deja acest principiu cu librăria EasyRDF din PHP. Din păcate în versiunea rdflib curentă (4.2.2) interogarea returnează un vector de tripleţi şi trebuie procesată ca atare. Poate fi convertit şi în graf cu puţin efort suplimentar, însă vom procesa rezultatul direct ca pe un vector Python, ţinând cont că fiecare element al vectorului e un tuplu cu 3 elemente:

- din fiecare afirmaţie ne interesează doar primul (subiectul, poziţia [0]) şi al treilea element (obiectul, poziţia [2]);
- ambele elemente le trecem prin funcţia qname() pentru a păstra doar varianta scurtă a termenului (de care avem nevoie în codul distilabil).

```
>>> knownpersonmappings=[(g.qname(x[0]),g.qname(x[2])) for x in result]
>>> knownpersonmappings
[('x:Mary', 'x:Anna'), ('x:Mary', 'x:Andrew'), ('x:Anna', 'x:Mary'), ('x:Anna', 'x:Andrew'), ('x:Andrew', 'x:Anna'), ('x:George', 'x:Andrew'), ('x:Andrew', 'x:Mary'), ('x:Andrew', 'x:George')]
```

Mai departe construim un schelet de dicţionar, în care fiecărui subiect îi este ataşat o listă goală. Deoarece dicţionarele nu permit repetarea cheilor, avem şi garanţia că subiectele extrase nu se vor repeta:

```
>>> subjects={sub:[] for (sub,ob) in knownpersonmappings}
>>> subjects
{'x:Mary':[], 'x:Anna':[], 'x:Andrew': [], 'x:George': []}
```

Apoi, pentru fiecare subiect unic (din *subjects*) îi construim şi ataşăm lista de persoane cunoscute (din *knownpersonmappings*).

```
>>> relationdict={sub:[b for (a,b) in knownpersonmappings if a==sub] for sub in subjects}
>>> relationdict
{'x:Mary': ['x:Anna', 'x:Andrew'], 'x:Anna': ['x:Mary', 'x:Andrew'], 'x:Andrew': ['x:Anna', 'x:Mary', 'x:George'], 'x:George': ['x:Andrew']}
```

Există şi un mod mai concis de a construi acest dicţionar, folosind funcţia setdefault():

```
>>>relationdict={}
>>> for (sub,ob) in knownpersonmappings:
        relationdict.setdefault(sub,[]).append(ob)
```

(se adaugă la dicţionar câte o listă goală pentru subiectele care nu există deja şi se adaugă obiectul la listă pentru cheile care există deja).

Acum avem toată procedura necesară construirii pas cu pas a celor două dicţionare, care în exemplul anterior fuseseră iniţializate static. Punem laolaltă toţi aceşti paşi, într-o pagină Web Python (salvaţi în fişierul *dynamicdata.py*):

```
import cherrypy
import mako.template
import rdflib
from rdflib.namespace import FOAF

class Site:
    @cherrypy.expose
    def index(self):
        g=rdflib.Graph()
        g.parse("http://localhost:8080/rdf4j-server/repositories/social/statements")
        namemappings=g[:FOAF.name:]
        namedict={g.qname(x):str(y) for (x,y) in namemappings}
        myquery="construct {?a foaf:knows ?b.?b foaf:knows ?a} where {?a foaf:knows ?b}"
        result=g.query(myquery)
        knownpersonmappings=[(g.qname(x[0]),g.qname(x[2])) for x in result]
        subjects={subj:[] for (subj,obj) in knownpersonmappings}
        relationdict={subj:[b for (a,b) in knownpersonmappings if a==subj] for subj in subjects}
        prefixdefinition="x: http://expl.at#"
        frontend=mako.template.Template(filename="table.txt")
        return frontend.render(names=namedict,relations=relationdict,pref=prefixdefinition)

cherrypy.quickstart(Site(),config="config.txt")
```

Șablonul Mako creat anterior, *table.txt*, poate să rămână același deoarece va primi aceleași informaţii, de inserat în acelaşi mod în acelaşi tabel HTML. Lansați noul site web prin tastarea în linia de comandă Windows (după ce aţi închis site-ul anterior cu Ctrl+C):

```
C:\pysite> dynamicdata.py
```

Ar trebui să aveți același rezultat ca în exercițiul anterior (atât în browser cât și codul sursă distilabil al paginii).

Diferența este că acum pagina Web va reacționa la schimbările din RDF4J. Pentru a verifica acest lucru, executați în RDF4J următoarea interogare de scriere:

```
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX x:<http://expl.at#>
insert data
{
x:Marianne foaf:name "Marianne Smith"; foaf:knows x:George, x:Anna
}
```

Observație: Nu încărcați un individ nou fără a-i devini atât eticheta foaf:name cât și relațiile foaf:knows, căci șablonul front-end a fost programat pe presupunerea că ambele există la fiecare individ (puteți aplica modificări pentru a permite absența uneia dintre acestea).

Dați un Refresh în fereastra browser-ului şi numele Marianne ar trebui să apară în tabel, iar dacă verificaţi codul HTML, veţi remarca şi relaţiile sale sociale cu George şi Anna:



Figura 47 Actualizarea paginii cu cunoștințe din RDF4J

```
<td style="color:red">Name</td></tr>
    <tr>
    <td about="x:George">
    <span property="foaf:name">George Smith</span>
    <span rel="foaf:knows">
        <span resource="x:Andrew"></span>
        <span resource="x:Marianne"></span>
    </span>
</td>
</tr>
    <tr>
    <td about="x:Marianne">
    <span property="foaf:name">Marianne Smith</span>
    <span rel="foaf:knows">
        <span resource="x:Anna"></span>
        <span resource="x:George"></span>
    </span>
```

Figura 48 Codul sursă a paginii web actualizate

Creați un serviciu endpoint care oferă tot conținutul bazei de cunoștințe în sintaxa solicitată de client

În continuare construim o pagină Web care oferă tot conţinutul bazei de cunoştinţe social unui client, în formatul solicitat de acesta, dintre următoarele: RDF/XML, N-triples, Turtle. Pentru orice alt format, se va returna un mesaj de eroare. Clientul îşi va anunţa formatul dorit cu ajutorul antetului HTTP Accept.

Creați următorul site (salvați-l în directorul C:\pysite cu numele *endpoint.py*):

```
import cherrypy
import rdflib

class Site:
  @cherrypy.expose
  def index(self):
    g=rdflib.Graph()
    g.parse("http://localhost:8080/rdf4j-server/repositories/social/statements")
    if ("Accept" in cherrypy.request.headers):
      if (cherrypy.request.headers["Accept"]=="application/rdf+xml"):
        return g.serialize(None,format="xml")
      elif (cherrypy.request.headers["Accept"]=="text/plain"):
        return g.serialize(None,format="nt")
      elif (cherrypy.request.headers["Accept"]=="text/turtle"):
        return g.serialize(None,format="turtle")
      else:
        return "your desired format is not available on this endpoint"
cherrypy.quickstart(Site(),config="config.txt")
```

Observații:

- Site-ul folosește metoda parse() pentru a extrage conținutul bazei de cunoștințe *social* într-un graf RDFLib*;*
- Dacă site-ul este accesat prin browser se va primi mesajul ”your desired format is not available...”, deoarece browserele setează antetul HTTP pe „html”, sintaxă care nu e prevăzută în lista de variante acceptate. Am putea combina acest exemplu cu exercițiul anterior pentru ca în caz de acces prim browser să se genereze o pagină HTML+RDFa!
- Dacă site-ul este accesat printr-o cerere HTTP care solicită sintaxa dorită cu antetul Accept va construi din graf un fişier text (cu serialize) în formatul solicitat şi îl va returna ca string; acest mecanism însă se aplică DOAR dacă se solicită una din sintaxele XML, N-triples sau Turtle (vezi tipurile MIME corespunzătoare lor);
- Nu este nevoie să folosim șabloane Mako deoarece site-ul nu este menit să fie vizitat de oameni, deci nu are cod HTML. El funcţionează ca un **serviciu** public („endpoint”) care răspunde cu grafuri în diverse sintaxe. Browserul va vedea doar acel mesaj de eroare.

Lansați noul site web prin tastarea în linia de comandă Windows (închideți site-ul anterior cu Ctrl+C):
```
C:\pysite> endpoint.py
```

Încercaţi să vizitaţi pagina cu browserul, la localhost:9999. Veţi vedea mesajul de eroare.

Încercați să accesaţi aceeaşi pagină cu o cerere HTTP configurată în Python în aşa manieră încât să se specifice antetul Accept, cu unul din formatele acceptate (rdf+xml de exemplu). Cererile HTTP în Python se pot construi cu ajutorul librăriei urllib ce oferă clasa Request:
```
>>> from urllib.request import *
>>> myrequest=Request("http://localhost:9999",headers={"Accept":"application/rdf+xml"},method="GET")
>>> temp=urlopen(myrequest)
>>> print(temp.read().decode())
```
(parametrul method poate lipsi în acest exemplu pentru că metoda GET e implicită)

În urma afişării, ar trebui să vedeţi graful sintaxa RDF/XML. Eventual, cu parse ar putea fi preluat în Python pentru alte prelucrări ulterioare la nivelul clientului Python.

Puteţi testa şi cu Postman cele 4 variante de răspuns programate.

Python 3 mai dispune şi de o librărie requests ce necesită instalare şi care oferă facilităţi suplimentare în construirea cererilor HTTP. Dacă aveţi Python 2, acolo există două librării diferite – urllib şi urllib2, care au fost unificate de Python 3 în urllib.

Creați un serviciu endpoint care oferă, în sintaxa dorită, toate afirmațiile despre un anumit termen

Creați următorul site web CherryPy (salvați-l în C:\pysite cu numele *endpoint2.py*):

```
import cherrypy
from urllib.request import *
from urllib.parse import *
address="http://localhost:8080/rdf4j-server/repositories/social?"
provenance="http://expl.at#"

class Site:
   @cherrypy.expose
   def index(self, term):
      params=urlencode({"query":"describe <"+provenance+term+">"})
      target=address+params
      myrequest=Request(target)
      if ("Accept" in cherrypy.request.headers):
         myrequest.add_header("Accept",cherrypy.request.headers["Accept"])
      temp=urlopen(myrequest)
      results=temp.read()
      return results
cherrypy.quickstart(Site(),config="config.txt")
```

Diferențe față de exercițiul anterior:

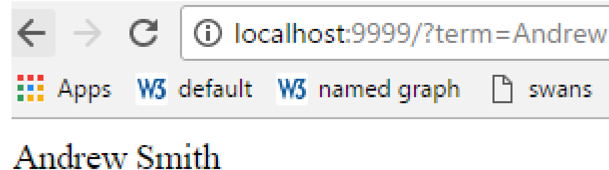
- De această dată pagina așteaptă un parametru GET numit "term". Valoarea sa va fi interpretată ca un termen RDF, pentru care se vor extrage din RDF4J toate afirmaţiile ce conţin acel termen cu ajutorul unei interogări DESCRIBE;
- De această dată decizia în legătură cu sintaxa răspunsului nu este luată în Python, de aceea lipseşte acea structură IF din exemplul precedent. Antetul Accept este forwardat către RDF4J (cu ajutorul add_header). RDF4J va lua decizia şi va returna graful gata serializat în format text, în sintaxa solicitată;
- Oricare ar fi rezultatele primite de la RDF4J, acestea sunt doar forwardate înapoi spre client fără nicio modificare.

În acest caz pagina Python nu face decât să intermedieze comunicarea între clienţi şi RDF4J. Aceasta e o practică uzuală atunci când nu dorim să facem cunoscută adresa serviciul de interogare RDF4J (de exemplu pentru a împiedica operaţii de scriere). Toate cererile spre RDF4J vor fi gestionate de pagina Python, care limitează accesul la baza de date la interogări DESCRIBE.

Lansați noul site web prin tastarea în linia de comandă Windows (închideți site-ul anterior cu Ctrl+C):

```
C:\pysite> endpoint2.py
```

Dacă doriți să îl accesați în fereastra browser-ului, este nevoie să se specifice parametrul term:



Deşi în browser se vede numele individului, puteţi verifica în codul sursă că browserul a primit cod RDF/XML (iar browserele afişează din orice cod XML doar nodurile text).

Pentru a primi afirmaţiile despre Andrew în alte sintaxe, trebuie să inițiem cereri HTTP cu valori Accept pe care RDF4J să le poată satisface. Precum în cazul precedent, exemplificăm cu o cerere trimisă din linia de comandă Python (dacă nu aţi închis IDLE nu mai repetaţi importul):

```
>>> from urllib.request import *
>>> myrequest=Request("http://localhost:9999?term=Mary",headers={"Accept":"application/x-turtle"})
>>> temp=urlopen(myrequest)
>>> print(temp.read().decode())
```

De această dată ar trebui să vedem afirmațiile despre Mary în format Turtle.

Acest exemplu este o implementare incompletă a noțiunii de **dereferențiere URI** („URI dereferencing”). O implementare completă a mecanismului de dereferenţiere ar trebui să permită ca resursa Andrew să poată fi solicitată prin adrese de forma *http://expl.at/Andrew* sau *http://expl.at#Andrew* în loc de folosirea unui parametru GET (*http://localhost:9999?term=Andrew*).

Ca sugestie, acest mecanism ar presupune următoarele funcționalități suplimentare:

- Definirea unui virtualhost pentru adresa *http://expl.at,* astfel încât serverul *(CherryPy)* să răspundă la acea adresă în loc de localhost;
- Programarea unui „dispatcher” *CherryPy* care să extragă numele resursei din adresa URL solicitată;
- Programarea unui script care să răspundă la solicitări dinspre browsere cu o pagină HTML ce prezintă unui vizitator uman afirmaţiile solicitate (vezi paginile DBPedia);
- Programarea paginii index astfel încât, în caz de solicitare de la un browser să redirecteze spre scriptul care oferă HTML (punctul precedent);
- Opțional, se poate adăuga ”negociere de conținut” (*content negotiation*) prin programarea paginii index în unul din comportamentele următoare:
  - Să răspundă la orice cerere cu o listă de sintaxe disponibile și apoi să aștepte o a doua cerere din partea clientului, care să specifice antetul Accept prin selecţie din lista oferită;
  - Să aştepte mai multe antete Accept cu priorități diferite și să aleagă sintaxa cea mai potrivită, conform unor reguli de potrivire între priorităţi şi sintaxele disponibile.

Conectarea la RDF4J cu ajutorul librăriei SPARQLWrapper

În exemplele cu generarea dinamică a paginii RDFa am folosit funcţia parse() oferită de clasa Graph din rdflib pentru a accesa grafuri de pe server. Funcţia parse() are unele limitări importante (similare cu utilizarea funcţiei load() din EasyRDF în PHP):

- Execută doar cereri GET

- Cererile sunt preconfigurate, nu putem să le configurăm noi. Putem doar să definim adresa ţintă, eventual să concatenăm parametri la aceasta, dar nu putem modifica metoda HTTP (POST, PUT, DELETE etc.) şi nici antetele HTTP (Accept, Content-Type).

Uneori e nevoie să depăşim astfel de obstacole pentru a avea un control mai fin asupra conexiunii cu serverul de grafuri. Avem la îndemână în Python soluţii similare cu cele oferite de EasyRDF în PHP, dar aici nu sunt toate disponibile în cadrul aceleiaşi librării:

- Folosim librăria SPARQLWrapper[59] dacă dorim să trimitem interogări SPARQL
- Folosim librăria urllib[60] dacă dorim să trimitem cereri HTTP configurare în cele mai mici detalii (ce pot şi ele să conţină interogări SPARQL)

Prima metodă e mai facilă când tot ce dorim e să interogăm. A doua metodă e mai puternică, permiţându-ne să dialogăm cu toate opţiunile pe care ni le permite interfaţa REST a serverului RDF4J (dar şi alte servere de grafuri).

Am văzut deja că în PHP avem clasa *EasyRdf_SPARQL_Client* pentru a trimite interogări SPARQL prin funcţii dedicate, la două adrese:

- *http://localhost:8080/rdf4j-server/repositories/databaseID* pentru operaţii de citire
- *http://localhost:8080/rdf4j-server/repositories/databaseID/statements* pentru operaţii de scriere

O clasă similară e disponibilă în Python, dar necesită instalarea librăriei externe *SPARQLWrapper*:

```
pip install sparqlwrapper
```

Presupunem că avem baza de date social anterior folosită. O vom accesa din linia de comandă Python:

```
>>> from SPARQLWrapper import SPARQLWrapper
>>> myclient=SPARQLWrapper("http://localhost:8080/rdf4j-server/repositories/social/statements")
>>> myquery="prefix : <http://expl.at#> insert data {:newgraph {:Endava a :Company. :Microsoft a :Company.}}"
>>> myclient.setQuery(myquery)
>>> myclient.query()
```

Am construit o interogare ce creează un graf nou şi adaugă nişte informaţii în acesta.

```
>>> myquery2="prefix : <http://expl.at#> with :newgraph delete {?x a :Company} insert {?x a :Organization} where {?x a :Company}"
>>> myclient.setQuery(myquery2)
>>> myclient.query()
```

Acum am executat un update asupra grafului nou, înlocuind tipul Company cu tipul Organization. Reamintim că SPARQL nu oferă o interogare UPDATE, ci folosim o combinaţie de DELETE şi INSERT (cu WITH pentru a limita modificarea la un anumit graf). De asemenea spre deosebire de PHP unde aveam funcţii diferite pentru citire şi scriere (query şi update) aici avem funcţia query pentru ambele tipuri de operaţii.

Mai departe executăm nişte operaţii de citire (pe care le-am putea realiza şi cu parse):

```
>>> from SPARQLWrapper import JSON
```

(am importat o constantă ce ne va permite să solicităm răspunsul în format JSON)

```
>>> readclient=SPARQLWrapper("http://localhost:8080/rdf4j-server/repositories/social")
>>> readquery="prefix foaf: <http://xmlns.com/foaf/0.1/> select ?subj ?name where {?subj foaf:name ?name}"
>>> readclient.setReturnFormat(JSON)
>>> readclient.setQuery(readquery)
>>> result=readclient.queryAndConvert()
>>> result
{'head': {'vars': ['subj', 'name']}, 'results': {'bindings': [{'name': {'type': 'literal', 'value': 'Mary Smith'}, 'subj': {'type': 'uri', 'value': 'http://expl.at#Mary'}},
{'name': {'type': 'literal', 'value': 'Andrew Smith'}, 'subj': {'type': 'uri', 'value': 'http://expl.at#Andrew'}}, {'name': {'type': 'literal', 'value': 'Anna Smith'},
'subj': {'type': 'uri', 'value': 'http://expl.at#Anna'}}, {'name': {'type': 'literal', 'value': 'George Smith'}, 'subj': {'type': 'uri', 'value': 'http://expl.at#George'}},
{'name': {'type': 'literal', 'value': 'Marianne Smith'}, 'subj': {'type': 'uri', 'value': 'http://expl.at#Marianne'}}]}}
```

[59] https://rdflib.github.io/sparqlwrapper/

[60] https://docs.python.org/3/library/urllib.html

(cu queryAndConvert() obţinem rezultatul în formatul solicitat, aici JSON; dacă folosim query(), rezultatul va fi un obiect de tip QueryResult, care poate fi ulterior convertit cu convert())

Trebuie să studiem structura JSON pentru a extrage date din ea. Următorul ciclu FOR va extrage perechi de subiecte şi numele lor:

```
>>> for x in results["results"]["bindings"]:
        print((x["subj"]["value"],x["name"]["value"]))
('http://expl.at#Mary', 'Mary Smith')
('http://expl.at#Andrew', 'Andrew Smith')
('http://expl.at#Anna', 'Anna Smith')
('http://expl.at#George', 'George Smith')
('http://expl.at#Marianne', 'Marianne Smith')
```

Dacă executăm interogări ce returnează grafuri complete (CONSTRUCT, DESCRIBE) atunci formatul solicitat ar trebui să fie TURTLE sau XML. În primul caz vom primi un string, în al doilea caz un obiect Graph conform cu librăria rdflib.

Exemplu:

```
>>> from SPARQLWrapper import XML, TURTLE
>>> readquery2="prefix foaf: <http://xmlns.com/foaf/0.1/> construct {?x foaf:knows ?y. ?y foaf:knows ?x} where {?x foaf:knows ?y}"
>>> readclient.setQuery(readquery2)
>>> readclient.setReturnFormat(XML)
>>> graphresult=readclient.queryAndConvert()
>>> print(graphresult.serialize(format="turtle").decode())
```

Observaţi că, dacă formatul solicitat e XML, queryAndConvert() creează o instanţă Graph pe care putem aplica funcţiile rdflib, de exemplu serialize(). Mai jos aplicăm şi alte tehnici rdflib suportate de clasa Graph, precum acea sintaxă specială de interogare prin g[s:p:o]:

```
>>> knownpairs={str(x[0]):str(x[2]) for x in graphresult[::]}
>>> knownpairs
```

(nu am mai făcut filtrare după foaf:knows, pentru că de asta s-a ocupat deja interogarea CONSTRUCT)

Dacă repetăm interogarea cu TURTLE ca format solicitat nu mai obţinem o instanţă Graph, ci un string pe care trebuie să îl convertim noi în obiect Graph:

```
>>> readclient.setReturnFormat(TURTLE)
>>> stringresult=readclient.queryAndConvert()
>>> print(stringresult.decode())
>>> g=rdflib.Graph()
>>> g.parse(data=stringresult,format="nt")
>>> print(g.serialize(format="nt").decode())
```

Procesarea JSON a rezultatelor SELECT cu SPARQLWrapper2

Pentru cazul frecvent al interogărilor SELECT cu răspuns în format JSON, avem şi o metodă mai facilă de a extrage datele, folosind versiunea 2 a constructorului SPARQLWrapper:

```
>>> from SPARQLWrapper import SPARQLWrapper2, JSON
>>> readclient=SPARQLWrapper2("http://localhost:8080/rdf4j-server/repositories/social")
>>> readquery="prefix foaf: <http://xmlns.com/foaf/0.1/> select ?subj ?name where {?subj foaf:name ?name}"
>>> readclient.setReturnFormat(JSON)
>>> readclient.setQuery(readquery)
>>> result=readclient.queryAndConvert()
```

Acum putem folosi funcţia getValues() ce ne oferă acces direct la valorile variabilelor returnate de SELECT, scutindu-ne de a naviga acea structură JSON:

```
>>> result.getValues("name")
[Value(literal:'Mary Smith'), Value(literal:'Andrew Smith'), Value(literal:'Anna Smith'), Value(literal:'George Smith'), Value(literal:'Marianne Smith')]
>>> result.getValues("name")[0].value
'Mary Smith'
>>> result.getValues("subj")
```

```
[Value(uri:'http://expl.at#Mary'), Value(uri:'http://expl.at#Andrew'), Value(uri:'http://expl.at#Anna'), Value(uri:'http://expl.at#George'),
Value(uri:'http://expl.at#Marianne')]
>>> result.getValues("subj")[0].value
'http://expl.at#Mary'
```

Construim lista numelor:

```
>>> namelist=[x.value for x in result.getValues("name")]
>>> namelist
```

În plus, putem procesa rezultatul ca pe un vector Python folosind:

```
>>> result.bindings
```

Dacă dorim să construim un dicţionar cu perechile {subiect, nume} vom executa:

```
>>> pairs={x["subj"].value:x["name"].value for x in result.bindings}
>>> pairs
{'http://expl.at#Mary': 'Mary Smith', 'http://expl.at#Andrew': 'Andrew Smith', 'http://expl.at#Anna': 'Anna Smith', 'http://expl.at#George': 'George
Smith', 'http://expl.at#Marianne': 'Marianne Smith'}
```

..sau, dacă dorim o listă de perechi:

```
>>> pairslist=[(x["subj"].value,x["name"].value) for x in result.bindings]
>>> pairslist
[('http://expl.at#Mary', 'Mary Smith'), ('http://expl.at#Andrew', 'Andrew Smith'), ('http://expl.at#Anna', 'Anna Smith'), ('http://expl.at#George', 'George
Smith'), ('http://expl.at#Marianne', 'Marianne Smith')]
```

Conectarea la RDF4J prin cereri HTTP configurate

După cum am văzut şi la PHP, putem oricând apela la librării HTTP cu ajutorul cărora să ne conectăm la serverul de grafuri prin intermediul interfeţei de tip REST pe care acesta o oferă. Când lucrăm cu librării HTTP putem configura toate detaliile unei cereri (vezi şi modul de utilizare a clasei EasyRDF_Http_Client în PHP). Python dispune de mai multe librării HTTP însă cea mai utilizată e urllib, aceasta fiind inclusă în instalarea de bază. Am folosit-o deja într-o formă simplă în exemplul cu construirea serviciului endpoint. În continuare vom exersa mai multe configurări.

Creaţi fişierul statements1.ttl în C:\pysite, cu următorul conţinut:

```
@prefix : <http://expl.at#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
:Anna foaf:knows :Andrew, :Alin, :Roxanne, :Maria.
:Andrew foaf:knows :Maria, :Jim.
:Jim foaf:knows :Ana, :Roxanne, :George.
:George foaf:knows :Jane.
```

Creaţi în RDF4J baza de date pythonrepo şi încărcaţi în ea următorul graf:

```
@prefix : <http://expl.at#>.
:friends {:Ana :cunoastePe :Andrei}
```

În continuare vom substitui conţinutul grafului :friends cu conţinutul fişierului statements1.ttl de pe disc. Pentru aceasta e nevoie de următoarele operaţii:

- vom accesa adresa la care RDF4J acceptă operaţii la nivel de graf (crearea, ştergerea sau înlocuirea unui graf întreg); adresa respectivă este
http://localhost:8080/rdf4j-server/repositories/pythonrepo/rdf-graphs/service?graph=.....
- vom încărca cererea HTTP cu fişierul citit de pe disc
- vom selecta metoda PUT care înlocuieşte un graf existent (POST adaugă conţinut la un graf existent)
- vom declara sintaxa în care trimitem datele (Turtle, cu codul MIME "application/x-turtle")

Începem prin a importa toate clasele necesare din librăria urllib:

```
>>> from urllib.parse import urlencode
>>> from urllib.request import Request, urlopen
```

Stocăm într-un string Python conţinutul fişierului de pe disc:

```
>>> fileobject=open("c:\pysite\statements1.ttl")
>>> content=fileobject.read().encode()
```

Construim adresa ţintă, concatenând identificatorul grafului (codificat). Observaţi că adresa la care trimitem operaţii la nivel de graf diferă de cea la care trimiteam interogări:

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo/rdf-graphs/service?"
>>> param=urlencode("graph":"http://expl.at#friends"})
>>> target=address+param
>>> target
'http://localhost:7200/repositories/pythonrepo/rdf-graphs/service?graph=http%3A%2F%2Fexpl.at%23friends'
```

Construim cererea cu ajutorul clasei Request. La momentul instanţierii indicăm adresa ţintă, datele pe care le trimitem, metoda HTTP (PUT) şi antetul Content-Type (sintaxa în care dorim să trimitem datele, aici Turtle):

```
>>> myrequest=Request(target,data=content,headers={"Content-Type":"application/x-turtle"},method="PUT")
```

În sfârşit, executăm cererea:

```
>>> urlopen(myrequest)
```

În continuare vom trimite o interogare INSERT la adresa de bază pentru operaţii de scriere:

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo/statements?"
>>> param=urlencode({"update":"prefix : <http://expl.at#> insert data {graph :other {:Ana :livesIn :Vienna}}"})
>>> target=address+param
>>> myrequest=Request(target,method="POST")
>>> urlopen(myrequest)
```

De data aceasta instanţierea cererii nu mai indică datele şi antetul HTTP Content-Type. Motivul este că datele trimise sunt cuprinse deja în interogare, nu sunt trimise prin corpul cererii HTTP. În consecinţă nici sintaxa datelor nu trebuie declarată. Verificaţi apariţia noului graf :other.

Mai departe vom solicita întreg conţinutul unui graf în format XML. Folosim adresa pentru operaţii la nivel de graf:

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo/rdf-graphs/service?"
>>> param=urlencode({"graph":"http://expl.at#friends"})
>>> target=address+param
>>> myrequest=Request(target,headers={"Accept":"application/rdf+xml"})
>>> temp=urlopen(myrequest)
>>> print(temp.read().decode())
```

Observaţii că în instanţierea Request() nu avem date de trimis, nu avem metodă de declarat (implicit se va folosi metoda GET), în schimb e folosit antetul Accept, prin care indicăm formatul în care dorim să primim datele (sintaxa RDF/XML). Deoarece rezultatul va fi un graf întreg îl putem încărca într-o instanţă a clasei Graph din rdflib pentru a fi procesat local prin funcţiile declarate deja.

Mai departe citim conţinutul unui graf filtrat după subiect. Am putea folosi o interogare SELECT, dar interfaţa REST a serverului acceptă şi unii parametri ce permit deghizarea de interogări în configurări HTTP: un parametru subj pentru filtrare după subiect, un parametru context pentru filtrare după graf, aplicate la adresa ce returnează conţinutul complet al bazei de date. Solicităm răspunsul în format N-triples:

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo/statements?"
>>> params=urlencode({"context":"<http://expl.at#friends>","subj":"<http://expl.at#Anna>"})
>>> target=address+params
>>> myrequest=Request(target,headers={"Accept":"text/plain"})
```

```
>>> temp=urlopen(myrequest)
>>> print(temp.read().decode())
```

Mai departe trimitem o interogare SELECT la adresa pentru operaţii de citire, solicitând răspunsul în format JSON pentru a fi prelucrat prin librăria json.

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo?"
>>> params=urlencode({"query":"prefix : <http://expl.at#> select * where {?x ?y ?z}"})
>>> target=address+params
>>> myrequest=Request(target,headers={"Accept":"application/sparql-results+json"})
>>> temp=urlopen(myrequest)
>>> results=temp.read().decode()
>>> print(results)
>>> import json
>>> dict=json.loads(results)
>>> dict["results"]["bindings"][0]
```

(afişăm prima înregistrare)

```
>>> subjects=[rec["x"]["value"] for rec in dict["results"]["bindings"]]
>>> subjects
```

(construim un vector cu toate subiectele din structura JSON)

Mai departe ştergem graful *:other* trimiţând o interogare DROP la adresa ce acceptă operaţii de scriere, prin metoda POST şi parametrul update:

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo/statements?"
>>> params=urlencode({"update":"drop graph <http://expl.at#other>"})
>>> target=address+params
>>> myrequest=Request(target,method="POST")
>>> urlopen(myrequest)
```

Mai departe ştergem graful *:friends* trimiţând o cerere HTTP de tip DELETE la adresa ce acceptă operaţii la nivel de graf:

```
>>> address="http://localhost:8080/rdf4j-server/repositories/pythonrepo/rdf-graphs/service?"
>>> params=urlencode({"graph":"http://expl.at#friends"})
>>> target=address+params
>>> myrequest=Request(target,method="DELETE")
>>> urlopen(myrequest)
```

7. Inferențe RDF Schema

Am văzut cum putem executa reguli încorporate în interogări (CONSTRUCT, INSERT WHERE) pentru a genera afirmații noi pornind de la cele existente.

O altă abordare pentru generarea de afirmaţii este stocarea regulilor direct în baza de cunoștințe, cu o serie de avantaje:

- Regulile permanent stocate în baza de cunoștințe sunt disponibile oricărui client, nu doar celui care realizează o anumită interogare
- Regulile stocate devin ele însele interogabile, se pot copia, importa deci reutiliza şi de alte baze de cunoștințe
- Orice modificare a datelor va provoca reexecutarea automată a regulilor pentru a menţine toată informaţia într-o formă consistentă (nu trebuie ca utilizatorul să se ocupe de reexecutarea unei interogări de câte ori se modifică datele).

Această abordare necesită însă unele elemente suplimentare față de modul de lucru de până acum:

1. Regulile trebuie scrise
    a. *fie într-o sintaxă dedicată pentru reguli (SPIN, SWRL etc.)*, fie
    b. *fie ca axiome* (afirmații RDF ce folosesc termeni standardizați de tip RDFS și OWL și au ca subiecte clase sau proprietăți); vom exemplifica doar abordarea bazată pe **axiome** (axiomele formează un **vocabular** sau o **ontologie**).
2. Serverul trebuie să aibă instalat un **motor inferențial** care să asigure generarea automată a noilor afirmații, fără intervenția utilizatorului. RDF4J oferă un motor inferenţial pentru reguli SPIN şi unul pentru axime RDFS. Îl vom exemplifica doar pe al doilea.

Creați un nou depozit de cunoștințe (cu numele InferRepo). De această dată selectați pentru tipul bazei de cunoștințe ”Native Java Store with RDF Schema” (în acest fel se va activa motorul inferenţial RDFS).

Executați o interogare generică:

```
select * where {?x ?y ?z}
```

Observați că deși nu am adăugat încă afirmații, baza de cunoștințe nu este goală! Aceasta conține deja 141 de axiome RDF Schema (**axiome implicite**) care stabilesc semnificația termenilor cheie din terminologia RDFS. Puteți să folosiți și opțiunea *Types* pentru a vedea o listă cu clase RDFS predefinite.

Pentru evita afișarea de URI integrali, definiți prefixele standard în ecranul *Namespaces*:

- rdf -> http://www.w3.org/1999/02/22-rdf-syntax-ns#
- rdfs -> http://www.w3.org/2000/01/rdf-schema#

Executați din nou interogarea pentru a vedea identificatorii standardizați. Efectuați clic pe rdfs:Resource pentru a vedea subclasele acesteia, proprietățile și axiomele unde apare (reamintim rdfs:Resource este clasa universală, căreia aparțin toate "resursele").

Faceți clic pe o proprietate pentru a vedea domeniul acesteia și codomeniul.

Resource: rdfs:Resource

Results per page: 200

Results offset: Previous 200 Next 200

Show data types & language tags: ☑

Sub Classes
- rdf:Alt
- rdf:Bag
- rdf:List
- rdf:Property
- rdf:Seq
- rdf:Statement
- rdf:XMLLiteral
- rdfs:Class
- rdfs:Container
- rdfs:ContainerMembershipProperty
- rdfs:Datatype
- rdfs:Literal

Properties
- rdf:type
- rdf:value
- rdfs:comment
- rdfs:isDefinedBy
- rdfs:label
- rdfs:member
- rdfs:seeAlso

Subject	Predicate	Object	Context
rdfs:Resource	rdf:type	rdfs:Resource	
rdfs:Resource	rdf:type	rdfs:Class	
rdfs:Resource	rdfs:subClassOf	rdfs:Resource	
rdf:type	rdf:type	rdfs:Resource	
rdf:type	rdfs:domain	rdfs:Resource	
rdf:Property	rdf:type	rdfs:Resource	
rdf:Property	rdfs:subClassOf	rdfs:Resource	

Figura 49 Subclasele, proprietățile axiomele pentru rdfs:Resource

Adăugați următoarea afirmație:

```
@prefix :<http://expl.at#>.
:JamesCameron :directorOf :Terminator.
```

Executați din nou interogarea generică. Observați că deși am adăugat doar o singură afirmație numărul total a crescut de la 141 la 147. Noile afirmații adăugate pot fi văzute în partea de jos a listei:

<http://expl.at#JamesCameron>	rdf:type	rdfs:Resource
<http://expl.at#JamesCameron>	<http://expl.at#directorOf>	<http://expl.at#Terminator>
<http://expl.at#directorOf>	rdf:type	rdf:Property
<http://expl.at#directorOf>	rdf:type	rdfs:Resource
<http://expl.at#directorOf>	rdfs:subPropertyOf	<http://expl.at#directorOf>
<http://expl.at#Terminator>	rdf:type	rdfs:Resource

Figura 50 Cunoțtințele generate la introducerea unei afirmații

- afirmația adăugată este *:JamesCameron :directorOf :Terminator*; celelalte au fost generate pe baza axiomelor RDF Schema după cum urmează:
    - pentru fiecare termen X a fost generată o afirmație *X a rdfs:Resource* (sunt 3 astfel de afirmații)
    - pentru orice termen X din mijlocul unei afirmații s-a generat o afirmație ce declară că este o proprietate: *X a rdf:Property*
    - în final, pentru proprietate s-a generat și afirmația că este propria subproprietate (*:directorOf rdfs:subPropertyOf :directorOf*)

Adăugați următoarele două afirmații:

```
@prefix :<http://expl.at#>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
```

```
:directorOf rdfs:domain :Director; rdfs:range  :Movie.
```

Spre deosebire de primul upload, aceste afirmații ies în evidență prin următoarele:

- au ca subiect o proprietate (:directorOf) și nu un individ
- folosesc termeni RDFS standard pentru a defini semnificația proprietății (faptul că :directorOf este o relație ce are loc între regizori și filme)

Astfel de afirmații poartă numele de **axiome**, iar mai multe astfel de axiome formează un **vocabular** (sau ontologie, dacă se folosesc și termeni OWL). Axiomele sunt interpretate ca reguli și permit motorului inferențial să realizeze deducții automate. Putem înțelege avantajele axiomelor de mai sus dacă executăm interogarea:

```
prefix : <http://expl.at#>
select * where {:Terminator a ?z}
```

Chiar dacă noi nu am adăugat o afirmație de forma ":Terminator a ...", RDF4J e capabil să răspundă că Terminator e un film (apartenența la clasa :Movie e dedusă din axioma cu rdfs:range), iar pe langă asta este și o resursă (deoarece orice e resursă). Dacă dorim să evităm răspunsul rdfs:Resource, putem include un filtru care să elimine din răspuns orice termen standardizat (care începe cu adresa W3C).

```
prefix : <http://expl.at#>
select * where
{
:Terminator a ?z
filter (!strstarts(str(?z),"http://www.w3.org"))
}
```

În mod similar putem întreba ce este :JamesCameron...

```
prefix : <http://expl.at#>
select * where {:JamesCameron a ?z}
```

...și vom afla că e regizor, deși noi nu am introdus o astfel de afirmație (ea e dedusă/generată din axioma cu rdfs:domain).

Aceste deducții pot fi verificate dacă executăm interogarea generică pentru a vedea ce afirmații noi au apărut în urma ultimului upload. Numărul total de afirmații a crescut la 159 (deci încă 12 noi):

- 2 dintre acestea sunt axiomele nou adăugate
- 2 dintre acestea sunt declarații de resurse noi (:Director a rdfs:Resource, :Movie a rdfs:Resource)
- 2 dintre acestea sunt declarații de clasă (:Director a rdfs:Class, :Movie a rdfs:Class) rdfs:domain și rdfs:range leagă întotdeauna o relație de clase (s-ar fi generat și declarația :directorOf a rdf:Property dacă nu exista deja)
- Fiecare clasă este o subclasă a ei însăși (:Director rdfs:subClassOf :Director, :Movie rdfs:subClassOf :Movie)
- Fiecare clasă este inclusă în rdfs:Resource (:Movie rdfs:subClassOf rdfs:Resource, :Director rdfs:subClassOf rdfs:Resource)
- Fiecare subiect din relația :directorOf este de tip :Director (JamesCameron a :Director)
- Fiecare obiect din relația :directorOf este de tip :Movie (:Terminator a :Movie)

<http://expl.at#JamesCameron>	rdf:type	rdfs:Resource
<http://expl.at#JamesCameron>	rdf:type	<http://expl.at#Director>
<http://expl.at#JamesCameron>	<http://expl.at#directorOf>	<http://expl.at#Terminator>
<http://expl.at#directorOf>	rdf:type	rdf:Property
<http://expl.at#directorOf>	rdf:type	rdfs:Resource
<http://expl.at#directorOf>	rdfs:domain	<http://expl.at#Director>
<http://expl.at#directorOf>	rdfs:range	<http://expl.at#Movie>
<http://expl.at#directorOf>	rdfs:subPropertyOf	<http://expl.at#directorOf>
<http://expl.at#Terminator>	rdf:type	rdfs:Resource
<http://expl.at#Terminator>	rdf:type	<http://expl.at#Movie>
<http://expl.at#Director>	rdf:type	rdfs:Resource
<http://expl.at#Director>	rdf:type	rdfs:Class
<http://expl.at#Director>	rdfs:subClassOf	rdfs:Resource
<http://expl.at#Director>	rdfs:subClassOf	<http://expl.at#Director>
<http://expl.at#Movie>	rdf:type	rdfs:Resource
<http://expl.at#Movie>	rdf:type	rdfs:Class
<http://expl.at#Movie>	rdfs:subClassOf	rdfs:Resource
<http://expl.at#Movie>	rdfs:subClassOf	<http://expl.at#Movie>

Figura 51 Cunoștințele generate în urma adăugării de afirmații legate de domeniul și codomeniul unei proprietăți

Să vedem cum se comportă baza de cunoștințe dacă ulterior introducerii axiomei se adaugă afirmații noi. Adăugați următoarele afirmații:

```
@prefix :<http://expl.at#>.
:TimBurton :directorOf :Batman.
:StevenSpielberg :directorOf :ET.
```

Apoi executați interogări precum "Ce este TimBurton?", "Ce este ET?" etc. și veți remarca faptul că răspunsurile există deja, ele fiind generate automat de motorul inferențial când s-au încărcat noile afirmații. Cu alte cuvinte, axiomele se introduc o singură dată și vor reexecuta deducţiile pentru toate afirmațiile care se adaugă ulterior.

Acesta e un avantaj major față de generarea cu INSERT sau CONSTRUCT, care generează afirmații noi DOAR pe baza afirmațiilor care există **la momentul interogării** (iar dacă ulterior se adaugă afirmații, interogarea trebuie reexecutată manual). În schimb motorul inferențial ține evidența generărilor care s-au făcut în trecut (are axiomele/regulile stocate permanent) și le recalculează la fiecare modificare (inclusiv la ștergeri, după cum se va arăta imediat).

Cu alte cuvinte, un motor inferențial **garantează la orice moment consistența logică a cunoștințelor generate** (în timp ce INSERT/CONSTRUCT o garantează cel mult la momentul executării interogării[61], pentru că interogările SPARQL nu "țin minte" ce interogări s-au mai executat înainte).

Am văzut cum funcționează axiomele ce folosesc predicatele standard rdfs:domain (domeniu) și rdfs:range (codomeniu). În continuare vom studia și alte axiome, cu alți termeni standard.

Adăugați următoarea afirmație:

```
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix :<http://expl.at#>.
:Director rdfs:subClassOf :Artist.
```

Această axiomă declară incluziunea între două clase. Cu alte cuvinte "oricine e regizor, este și artist". Acum dacă întrebăm "ce este :JamesCameron?"...

[61] uneori nici atunci, dacă de la INSERT-ul precedent au avut loc și ștergeri

```
prefix : <http://expl.at#>
select * where {:JamesCameron a ?z}
```

...observăm că acesta a devenit și Artist deși, la fel ca în cazul precedent, nu am introdus nicio afirmație de forma :JamesCameron a :Artist. Această afirmație s-a dedus din axioma de incluziune.

Mai multe axiome de tip rdfs:subClassOf pot alcătui o ierarhie de clase incluse unele în altele ("taxonomie"). O astfel de ierarhie permite ca un individ să primească mai multe tipuri (deci pot exista mai multe răspunsuri la întrebarea "Ce este X?", răspunsurile fiind mai vagi sau mai precise în funcție de cât de jos/sus poate fi poziționat individul X în ierarhie). Ierarhia de clase este scheletul oricărui vocabular/ontologii și are în vârf clasa rdfs:Resource (clasa tuturor resurselor include orice altă clasă, deci la întrebarea "Ce este X?" va exista cel puțin răspunsul rdfs:Resource).

Executați din nou interogarea generică. S-au adăugat 6 afirmații noi:

- Axioma pe care am adăugat-o;
- Afirmația ce ne dă răspunsul nou la interogarea de mai sus (că JamesCameron este, pe lângă regizor, și artist);
- Declarația de resursă nou adăugată (:Artist a rdfs:Resource)
- Declarația de clasă nouă (:Artist a rdfs:Class)
- Orice clasă este inclusă în rdfs:Resource (:Artist rdfs:subClassOf rdfs:Resource)
- Orice clasă este propria subclasă (:Artist rdfs:subClassOf :Artist)

<http://expl.at#JamesCameron>	rdf:type	rdfs:Resource
<http://expl.at#JamesCameron>	rdf:type	<http://expl.at#Director>
<http://expl.at#JamesCameron>	rdf:type	<http://expl.at#Artist>
<http://expl.at#JamesCameron>	<http://expl.at#directorOf>	<http://expl.at#Terminator>
<http://expl.at#Director>	rdfs:subClassOf	<http://expl.at#Artist>
<http://expl.at#Movie>	rdf:type	rdfs:Resource
<http://expl.at#Movie>	rdf:type	rdfs:Class
<http://expl.at#Movie>	rdfs:subClassOf	rdfs:Resource
<http://expl.at#Movie>	rdfs:subClassOf	<http://expl.at#Movie>
<http://expl.at#Artist>	rdf:type	rdfs:Resource
<http://expl.at#Artist>	rdf:type	rdfs:Class
<http://expl.at#Artist>	rdfs:subClassOf	rdfs:Resource
<http://expl.at#Artist>	rdfs:subClassOf	<http://expl.at#Artist>

Figura 52 Cunoștințe generate de afirmații legate de ierarhia claselor

Adăugați următoarea afirmație:

```
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix :<http://expl.at#>.
:AmericanDirector rdfs:subClassOf :Director.
```

Întrebați "Ce este James Cameron?"

```
prefix : <http://expl.at#>
select * where {:JamesCameron a ?z}
```

Răspunsul e identic cu cazul precedent, deoarece noua axiomă adăugată, chiar dacă e tot o relație subClass, nu afectează tipul lui James Cameron (faptul că James Cameron e regizor nu e o garanție că e neapărat și regizor american)! Practic am introdus regula "oricine e regizor american e și regizor" însă regula nu e valabilă și în sens invers – faptul că JamesCameron e regizor nu îl face automat și regizor american!

Ne putem convinge de asta executând interogarea generică. Au apărut 6 noi afirmații, însă nu a apărut niciuna nouă despre James Cameron:

- Axioma adăugată de noi
- Declarația de resursă nouă (:AmericanDirector a rdfs:Resource)
- Declarația de clasă (:AmericanDirector a rdfs:Class)
- Orice clasă e inclusă în rdfs:Resource (:AmericanDirector rdfs:subClassOf rdfs:Resource)
- Orice clasă este propria subclasă (:AmericanDirector rdfs:subClassOf :AmericanDirector)
- Tranzitivitatea ierarhiei de clase: AmericanDirector devine și subclasă a clasei Artist, deoarece aceasta includea clasa Director (:AmericanDirector rdfs:subClassOf :Artist).

Observați că James Cameron nu a primit tipul AmericanDirector, deoarece relația rdf:type ("a") se propagă doar "în sus" prin ierarhia de clase (prin deducție logică, James Cameron primește doar tipurile de deasupra lui Director, dar nu și pe cele de dedesubt!) Sau, cu alte cuvinte, orice instanță a unei clase este și o instanță a superclaselor sale, dar nu neapărat și a subclaselor.

Să se adauge următoarea afirmație:

```
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix :<http://expl.at#>.
:directorOf rdfs:subPropertyOf :creatorOf.
```

Această axiomă are un rol similar cu axiomele subClass, dar se aplică asupra relațiilor și nu asupra claselor! Ea poate fi interpretată sub forma unei implicații logice între două relații, a doua fiind o versiune mai vagă a primeia: "Dacă x e regizorul lui y, înseamnă că x e și creatorul lui y".

Executați din nou interogarea generică pentru a vedea ce se deduce din ea. S-au adăugat 5 afirmații noi:

- Axioma nou introdusă
- Afirmația dedusă, că JamesCameron nu doar a regizat Terminator, ci a și creat Terminator (deci "a creat" e o versiune mai vagă a lui "a regizat"); atenție, deducția e aplicată relației, acțiunii de a regiza (:directorOf), nu clasei regizorilor (:Director)
- Declarația de resursă nouă pentru relație nou introdusă (:creatorOf a rdfs:Resource)
- Declarația că avem o nouă proprietate (:creatorOf a rdf:Property)
- Orice proprietate este propria subproprietate (:creatorOf rdfs:subPropertyOf :creatorOf)

<http://expl.at#JamesCameron>	<http://expl.at#directorOf>	<http://expl.at#Terminator>
<http://expl.at#JamesCameron>	<http://expl.at#creatorOf>	<http://expl.at#Terminator>
<http://expl.at#creatorOf>	rdf:type	rdf:Property
<http://expl.at#creatorOf>	rdf:type	rdfs:Resource
<http://expl.at#creatorOf>	rdfs:subPropertyOf	<http://expl.at#creatorOf>

Figura 53 Cunoștințe generate de ierarhia proprietăților

Acest exemplu ne demonstrează că nu doar clasele pot forma o ierarhie, ci și proprietățile! O supraproprietate e o relație aplicabilă între aceiași indivizi ca și subproprietățile sale; sau, cu alte cuvinte, o supraproprietate e o versiune mai vagă și mai general aplicabilă a subproprietăților sale (ex.: dacă cineva a regizat ceva, a și creat acel ceva; dacă cineva e frate cu cineva, e și rudă cu acel cineva etc.).

Același mecanism de deducție "în sus pe ierarhie", discutat deja la subClass, se aplică și aici. Adăugați următoarea afirmație:

```
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix :<http://expl.at#>.
:codirectorOf rdfs:subPropertyOf :directorOf.
```

Interpretare: "dacă x a co-regizat y, putem spune că x a regizat y".

Executați din nou interogarea generică. S-au adăugat 5 afirmații noi:

- Cea introdusă de noi
- Declarația de resursă pentru noua resursă (:codirectorOf a rdfs:Resource)
- Orice resursă implicată în relația rdfs:subPropertyOf este o proprietate (:codirectorOf a rdf:Property)
- Orice proprietate este o subproprietate a ei (:codirectorOf rdfs:subPropertyOf :codirectorOf)
- Afirmația dedusă că codirectorOf este de asemenea și subproprietate a relației creatorOf (:codirectorOf rdfs:subPropertyOf :creatorOf)

Observați că nu s-a generat afirmația :JamesCameron :codirectorOf :Terminator, deoarece deducția se poate face doar în sus (din existența subproprietății se deduce supraproprietatea dar nu și invers!)

Realizați o ștergere a unei axiome (declarația domeniului relației :directorOf):

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix : <http://expl.at#>
delete data {:directorOf rdfs:domain :Director}
```

Observați că s-au șters mai multe, nu doar o singură afirmație. Pe lângă cea ștearsă explicit, au dispărut și concluziile care s-au dedus din aceasta:

- :JamesCameron a :Director (concluzia dedusă din domeniul relației :directorOf)
- :JamesCameron a :Artist (dedusă din concluzia anterioară combinată cu subClassOf)

Prin această ștergere putem observa *unul din avantajele majore ale utilizării motoarelor inferențiale, comparativ cu regulile încorporate în interogări CONSTRUCT/INSERT: dacă baza de cunoștințe se modifică, toate deducțiile se reexecută automat pentru a garanta în orice moment consistența logică a afirmațiilor rămase*.

8. Crearea unui instrument de modelare conceptuală agilă

Modelarea conceptuală este o tehnică de reprezentare a cunoștințelor în formă grafică, menite să asigure suport decizional și analitic cu privire la sistemul modelat. Cele mai cunoscute instrumente de modelare sunt orientate pe descriere de sisteme softare (UML)[62], procese de afaceri (BPMN)[63] sau arhitecturi de întrepinderi (Archimate)[64]. Dincolo de standarde, o abordare recentă (Karagiannis, 2015) a propus noțiunea de **limbaje de modelare agile**, în care nu se respectă un anume standard ci se personalizează limbajul și instrumentul de modelare pentru cerințele specifice ale unor utilizatori sau domeniu de aplicare. Agilitatea se poate manifesta la nivel grafic, sintactic sau semantic.

Modelarea diagramatică asistată de calculator poate în prezent să beneficieze de o notație îmbogățită cu o varietate de caracteristici, precum: (i) interactivitate (simbolurile ar putea juca rolul unor hiperlegături) (ii) dinamicitate (simboluri care se schimbă în funcție de semantică sau de anumite proprietăți înțelese de mașini), (iii) semantică vizuală (informația comunicată prin aspecte ornamentale sau chiar animații). Niciuna din acestea nu ar fi fost posibile la folosirea creionului sau a șabloanelor iar varietatea de opțiuni disponibile în această privință este în prezent subiect al cerințelor de modelare, împreună cu alte nevoi de particularizare ale utilizatorilor în cauză (modelatorii) ce țin de (i) funcționalitatea bazată pe modele, (ii) acoperirea semantică a modelelor sau (iii) gradul specificității de domeniu.

Ingineria software a recunoscut natura dinamică a cerințelor și valoarea competitivă a unui răspuns flexibil la cerințele evolutive prin apariția Manifestului Agile[65] și a unei comunitătii întregi care a pus la îndoială metodele tradiționale rigide de dezvoltare a software-ului. Alte provocări similare legate de cerințele de modelare pot fi întâlnite în practica ingineriei metodei de modelare, în special în aplicarea modelării întreprinderii. În consecință, o abordare de inginerie agilă a metodei de modelare - **Agile Modelling Method Engineering** (AMME) și a facilitatorilor acesteia a fost propusă de (Karagiannis, 2015).

De-a lungul istoriei, ingineria software bazată pe modele a folosit modelele în mai multe modalități. Paradigma "modelarea e programare" folosea modelele pentru generarea de cod executabil. Această viziune fost ulterior completată de abordarea "modelarea este configurare" în care sistemele în momentul execuției erau parametrizate cu informații din modele iar modelele aveau rolul de "panouri de control" pentru a influența comportamentul din timpul execuției – de ex. în contextul sistemelor informatice ghidate de procese (van der Aalst, 2009) cu ajutorul serializării XML a modelelor. Pe măsură ce modelarea întreprinderii s-a extins mai departe de obiectivele managementului proceselor de afaceri, pentru construirea unei reprezentări holistice a întreprinderii, modelele diagramatice au devenit modalități de a reprezenta cunoștințe privind variate aspecte ale întreprinderii (de ex. procese, scopuri, capabilități), astfel că astăzi putem încadra modelarea conceptuală în clasa tehnologiilor semantice cu ajutorul cărora se pot construi sisteme bazate pe cunoștințe.

Limbajele agile de modelare se dezvoltă incremental și iterativ luând în considerare straturile de abstractizare ale metamodelării din Figura 54:

- la nivelul diagramelor, modelele se creează cu un anumit limbaj de modelare ale cărui notație, sintaxă și semantică pot fi personalizate agil pe nivelul superior;
- la nivelul Meta, terminologia și gramatica limbajului de modelare se adaptează cerințelor, incluzând concepte, simboluri grafice, proprietăți editabile, restricții sintactice și semantice ce trebuie implementate în softare (fără acestea, un instrument de modelare nu este altceva decât un instrument de desenare);
- la nivelul Meta-Meta sunt fixate conceptele primitive din care se instanțiază componentele limbajului de modelare; pentru a facilita implementarea agilă (incrementală și iterativă) a

[62] http://www.uml.org/
[63] http://www.bpmn.org/
[64] http://www.opengroup.org/subjectareas/enterprise/archimate-overview
[65] http://agilemanifesto.org/

instrumentului de modelare se folosesc medii de programare specializate ce oferă un nivel Meta-Meta predefinit și reutilizabil. Exemple de astfel de platforme sunt ADOxx[66], MetaEdit+[67], Eclipse Modeling Framework[68].

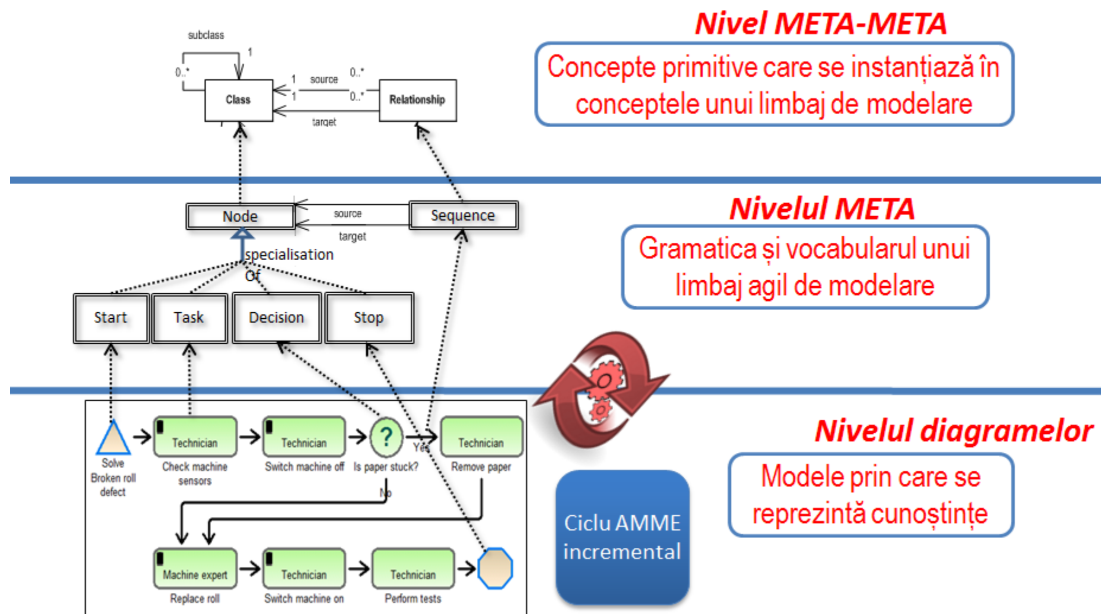


Figura 54 Straturile de abstractizare ale metamodelării

În abordarea noastră, aceste nivele pot fi mapate pe abstractizările standardului RDF – o sugestie în acest sens este dată în Figura 55.

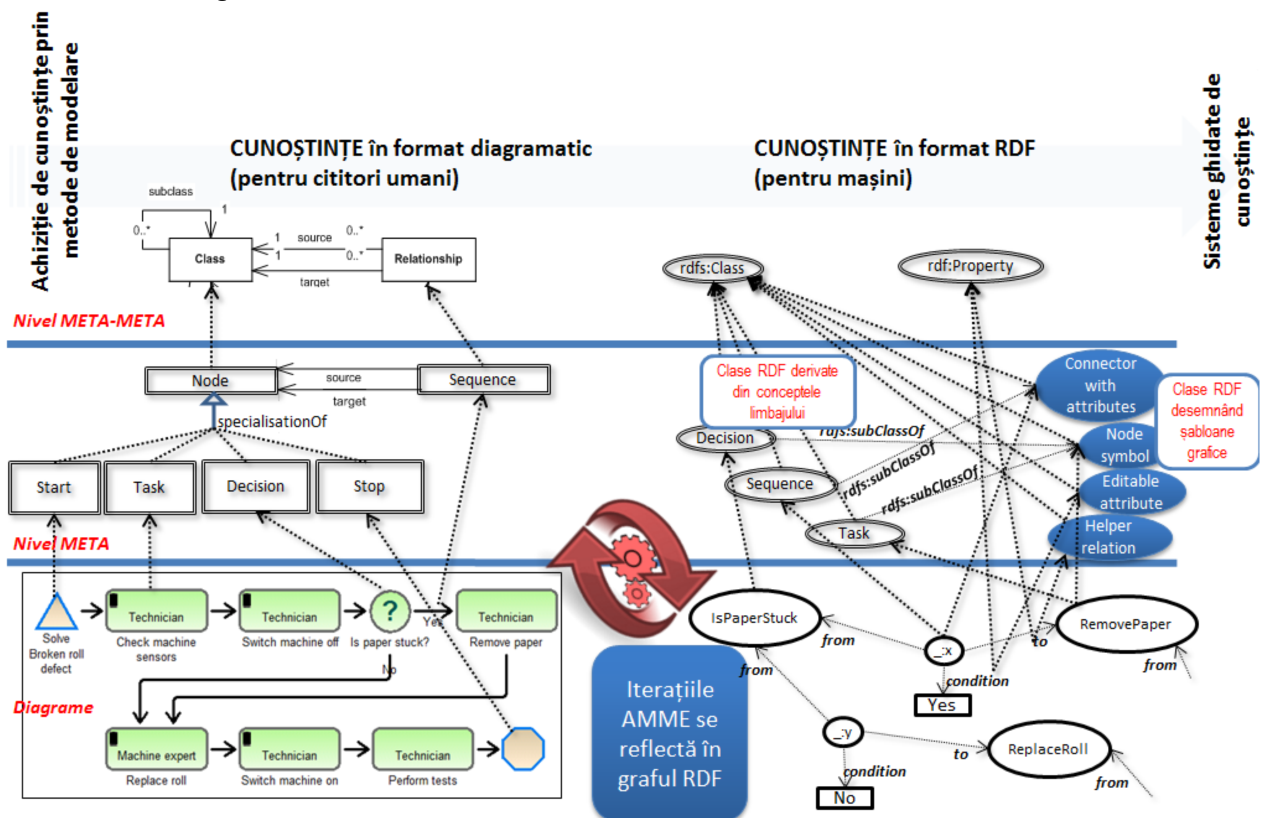


Figura 55 Mapare dintre straturile metamodelării și cele ale modelului RDF

66 https://www.adoxx.org/
67 https://www.metacase.com/products.html
68 https://www.eclipse.org/modeling/emf/

Mecanismul de export al diagramelor după aceste principii este disponibil în mai multe instrumente și forme: softwareul BEE-UP[69] oferă acest export pentru diagrame realizate în limbaje de modelare standard – UML, BPMN, EPC, Petri Nets, ER. Instrumentul este disponibil și ca plug-in pentru platforma ADOxx[70], caz în care exportul va putea fi adaptat agil la modificările aduse limbajului de modelare[71] .

În capitolele ce urmează vom exemplifica aceste tehnologii în două etape:

1. Mai întâi prezentăm un tutorial pentru crearea unui limbaj de modelare, modificarea sa agilă (evolutivă) și implementarea într-un software utilizabil cu ajutorul mediului de programare ADOxx;
2. Apoi prezentăm un tutorial privind exportul de diagrame în grafuri RDF în vederea expunerii acestora în format interpretabil către mașini. Se vor exemplifica și interogări de grafuri aplicate asupra informației de diagrame.

8.1 Implementarea unui limbaj agil de modelare

În exercițiile ce urmează vom defini un limbaj de modelare pentru a descrie rețete de gătit sub forma unor diagrame și îl vom implementa într-un software de modelare cu ajutorul mediului de programare ADOxx.

Vom porni ADOxx Development Toolkit (credențialele predefinite sunt user: “Admin”, parola: “password”) și apoi vom deschide fereastra Library Management.

1. Selectați tab-ul Settings. Acesta va afișa toate metodele de modelare create în instalarea curentă ADOxx – fiecare dintre ele este denumită “library” și fiecare va genera propriul instrument de modelare, corespunzător propriului limbaj de modelare. La început, după instalare, va fi doar una singură – Experimentation Library. Aceasta este șablon gol pentru o metodă de modelare care este inclus în ADOxx. Veți extinde acest “șablon gol pentru limbajul de modelare” prin adăugarea de concepte și simboluri proprii. Dacă se dorește păstrarea acestei versiuni nemodificate (pentru proiecte viitoare sau pentru a se permite revenirea lucrului la șablonul gol) ar trebui să se exporte:
    a. alegeți tab-ul Management
    b. selectați Experimentation Library
    c. exportați-o către un fișier ABL. Păstrați acest fișier pentru situații viitoare, în caz că veți avea nevoie să reluați de la început o nouă metodă de modelare (în acest caz ar trebui să o importați și să îi schimbați numele)
2. În următoarele exerciții vom lucra pe Experimention Library. Selectați Dynamic Library.
3. Select Class hierarchy pentru a vedea conceptele limbajului.

[69] http://austria.omilab.org/psm/content/bee-up/info
[70] https://www.adoxx.org/
[71] http://austria.omilab.org/psm/content/enterknow/info

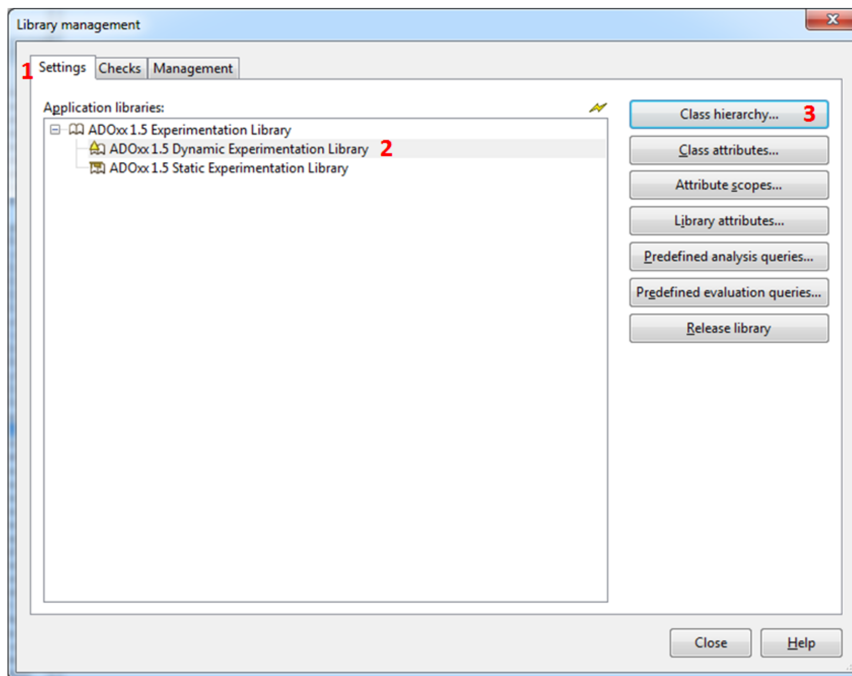


Figura 56 Fila Settings din fereastra Library Management

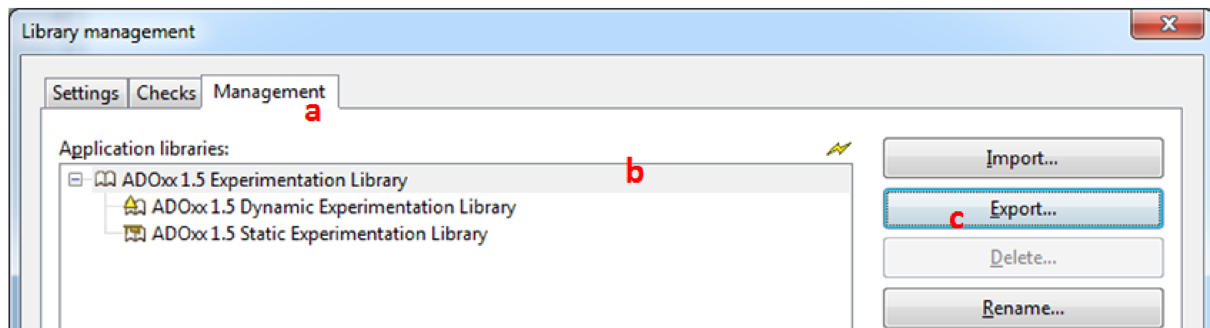


Figura 57 Fila Management din fereastra Library Management

În fereastra Class hierarchy urmați explicațiile numerotate de mai jos:

1. Selectați View – Metamodel, apoi View – Class hierarchy pentru a vedea întreaga ierarhie a conceptelor din ADOxx, inclusiv conceptele predefinite (cele care încep cu _D_). Este posibil să lucrați fără a vedea întreaga ierarhie, dacă moștenirea nu este relevantă (de exemplu dacă se creează un limbaj plat unde conceptele nu sunt moștenite unul din altul) sau dacă se dorește a se vedea doar conceptele din limbajul propriu și nu cele predefinite în ADOxx. Din acest motiv nu este vizibilă în mod implicit întreaga ierarhie;
2. Dacă ați ales să faceți vizibilă ierarhia, puteți observa că **rădăcina tuturor conceptelor este _D_construct,** care oferă structura cadru predefinită pentru toate conceptele. Orice concept pe care îl definiți pentru limbajul propriu va moșteni anumite proprietăți/metode predefinite din _D_construct;
3. În zona 3 din imaginea de mai jos se poate vedea din ce este compusă această structură cadru – câteva atribute predefinite care ne vor ajuta să definim notația, sintaxa și semantica pentru fiecare concept;
4. Faceți click-dreapta pe _D_construct și selectați New class pentru a defini un nou concept – Node. Acesta va fi poziționat imediat sub _D_construct în ierarhie;
5. Faceți click-dreapta pe Node, alegeți din nou New class și definiți un nou concept – CookingStep;
6. În zona 6 se pot vedea atributele pentru CookingStep care au fost moștenite de la Node;
7. În zona 7 se pot vedea atributele pentru Node moștenite de la _D_construct (a cărui atribute sunt vizibile în zona 3)
8. Faceți click-dreapta pe Node și creați 2 subconcepte adiționale: Start și Stop. Acestea de asemenea vor moșteni anumite atribute. Scopul acestor concepte este de a crea un limbaj de modelare care

ne va permite să descriem rețete de gătit sub formă de diagrame. O diagramă pentru o rețetă de gătit (Cooking Recipe) va fi o secvență formată din pași de gătit (Cooking Steps). În plus, conceptul Start va indica unde începe rețeta iar conceptul Stop va indica unde se termină. Toate aceste trei concepte sunt specializări ale conceptului Node.

9. Pe lângă Node, o diagramă pentru rețeta de gătit va avea conectori (săgeți) pentru a indica ordinea nodurilor. Aceste săgeți vor fi definite ca relații (mai târziu) după cum se văd în zona 9 din captura de ecran. Deocamdată, ne vom concentra asupra conceptului Node.

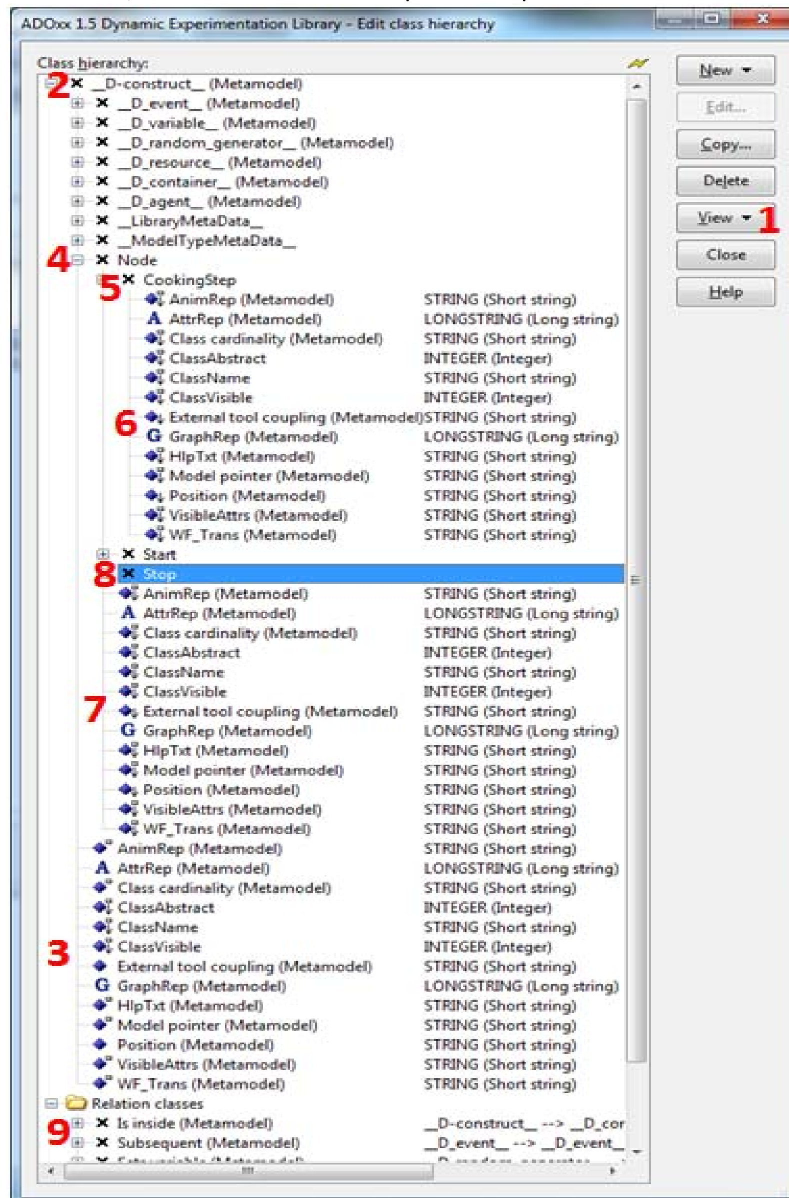


Figura 58 Ierarhia de clase din metoda de modelare

Majoritatea conceptelor dintr-un limbaj de modelare trebuie să aibă definit un simbol grafic pentru a le distinge de alte concepte și pentru a comunica un înțeles. Acest lucru se realizează în atributul predefinit GraphRep.

La deschiderea GraphRep trebuie să vă asigurați că este corespunzător pentru CookingStep și nu pentru Node sau _D_construct. Atributul GraphRep este moștenit de aceea este prezent la toate conceptele. Trebuie însă definit doar pentru acele concepte care vor fi prezente (instanțiate) în diagrame. Node și _D_construct sunt prezente în ierarhie doar pentru a facilita moștenirea – dar nu vor fi prezente în diagrame, de aceea nu au nevoie de a avea o notație grafică!

Editați atributul GraphRep pentru CookingStep pentru a defini simbolul său grafic. Se va afișa fereastra de mai jos:

1. apăsați butonul 1 pentru a deschide fereastra de definiție GraphRep
2. folosiți zona Text pentru a scrie codul GraphRep (puteți folosi butonul Help pentru a consulta sintaxa).

```
GRAPHREP
FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm
```

GraphRep este un limbaj grafic vectorial care ne permite să definim forme într-un sistem fix de coordonate, a cărui centru este punctul dat de click (unde se plasează simbolul pe planșa de modelare). Detalii despre exemplul din figură:

   a. cuvântul cheie GRAPHREP este obligatoriu la început
   b. comanda FILL va defini culoarea pentru umplerea formelor care vor urma (de aceea ordinea este importantă! dacă se specifică culoarea cu FILL după forme, nu se va aplica!)
   c. comanda ELLIPSE definește o elipsă cu raza indicată pentru axa x și y. Dacă sunt egale se va obține un cerc.

3. Folosiți butonul Paint pentru a obține o sugestie vizuală legată de cum va arată simbolul. Acesta va funcționa doar pentru definiții statice (în notațiile dinamice sau interactive, nu vor fi vizibile aici toate detaliile)
4. În zona 4 puteți observa efectul folosirii Paint.

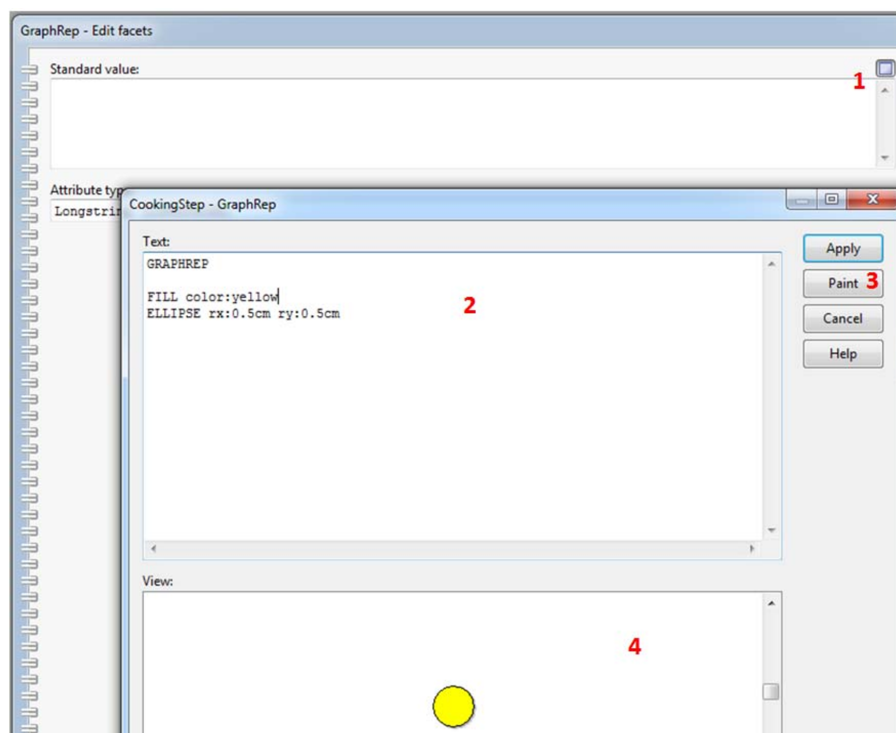


Figura 59 Fereastra GraphRep

Închideți fereastra GraphRep cu Apply și întoarceți-vă în ierarhia de clase. Creați o reprezentare grafică GaphRep pentru conceptul Start prin următorul cod:

```
GRAPHREP
TEXT "Start"
```

Această notație este minimalistă, nici măcar nu definește o formă grafică. Simbolul pentru Start va fi doar o etichetă text statică afișând cuvântul “Start”. De asemenea este posibil să se combine notația grafică cu astfel de etichete text pentru a se comunica înțelesul conceptelor la nivel vizual, într-un mod cât mai clar posibil.

Întoarceți-vă la ierarhia de clase și modificați în mod similar GraphRep pentru conceptul Stop:

```
GRAPHREP
```

```
TEXT "Stop"
```

Revenim la ierarhia de clase pentru a crea săgeata care va conecta pașii de gătit pentru a indica ordinea acestora. Aceasta este numită în ADOxx "clasă relație":

1. Faceți click-dreapta pe clasele Relation și alegeți New relationclass
2. Indicați numele relației (followed By) precum și sursa și destinația permisă (pentru ambele Node)

Ar trebui acum să devină mai clar de ce am creat conceptul intermediar Node (deși nu ar avea GraphRep => nu va apărea direct în diagrame): o relație în ADOxx poate avea doar un singur concept sursă și unul singur pentru destinație, astfel dacă dorim să permitem mai multe concepte să fie conectate cu aceeași relație trebuie să le unim într-o clasă abstractă (clasă abstractă = clasă care nu are GraphRep).

Cu ajutorul conectorului "followed By" vom putea conecta orice Node de orice Node (indiferent care ar fi subclasele instanțiate – Start, CookingStep, Stop). Mai târziu vom adăuga niște restricții – de exemplu pentru a interzice unei săgeți de a intra într-un element Start. A se observa că spre deosebire de concepte, relațiile nu sunt aranjate într-o ierarhie și din această cauză nu se poate realiza moștenire între relații.

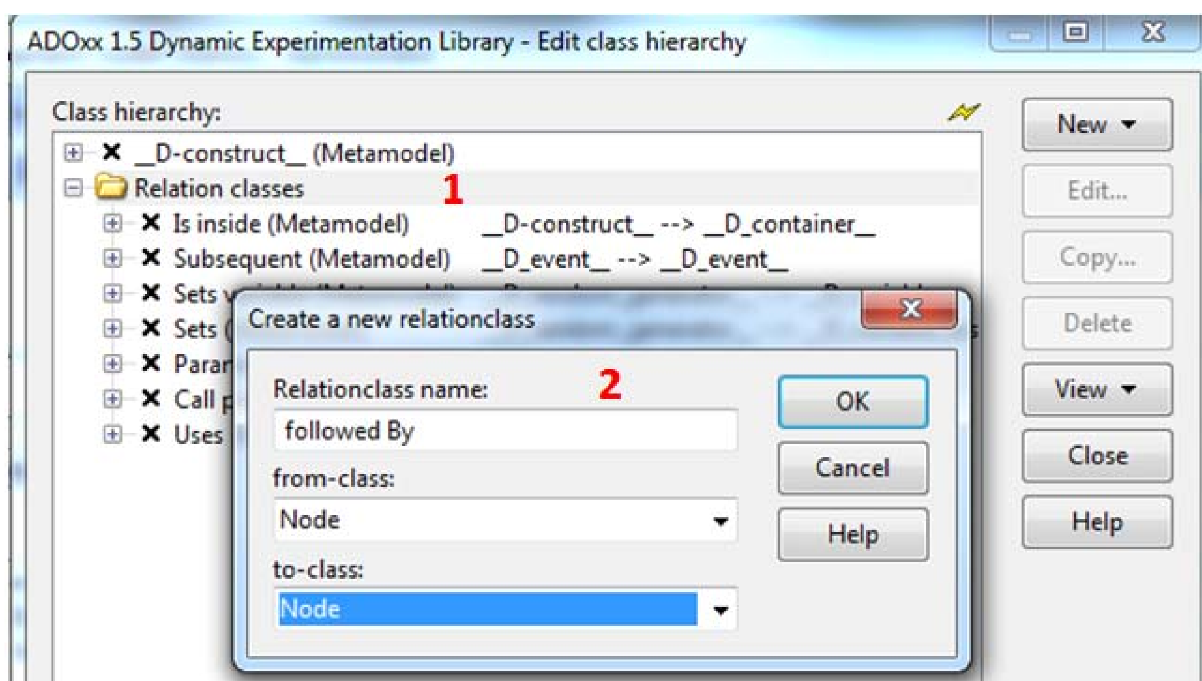


Figura 60 Clasele de relații din metoda de modelare

Observați că relațiile de asemenea au propriul GraphRep, care este definit cu aceeași sintaxă ca mai sus. PEN ne permite să definim linia (culoarea, lățimea, stilul etc.) și FILL ne permite să definim o culoare de umplere. Mai multe definiții PEN/FILL ar putea să modifice caracteristicile diverselor părți a unui simbol grafic.

Totuși există o diferență – pentru relații este nevoie a se defini 3 secțiuni a simbolului grafic:

```
GRAPHREP

PEN color:red w:0.1cm

EDGE

PEN color:blue

START
POLYGON 4 x1:-0.2cm y1:0.2cm x2:0.2cm y2:0.2cm x3:0.2cm y3:-0.2cm x4:-0.2cm y4:-0.2cm

END
```

POLYLINE 3 x1:-0.5cm y1:0.2cm x2:0cm y2:0cm x3:-0.5cm y3:-0.2cm

1. Linia este definită prin EDGE. În cazul nostru, prima comandă PEN se aplică acesteia și va specifica culoarea roșie și lățimea liniei. Se mai poate include aici o formă grafică sau un text, care are fi plasate în mijlocul liniei.
2. Începutul conectorului este definit cu START. În cazul nostru, acesta este un poligon pătrat cu 4 puncte. Coordonatele punctelor sunt măsurate în mod relativ cu punctul de începere a liniei. Ultima comandă PEN se aplică aici și specifică culoarea albastră.
3. Ultima parte este definită cu END. În cazul nostru, aceasta este o polilinie (care în contrast cu poligonul nu este închisă) cu 3 puncte, pentru a determina apariția unui vârf de săgeată. Centrul coordonatelor este unde se termină linia. Ultima comandă PEN se aplică din nou de aceea culoarea este albastră.

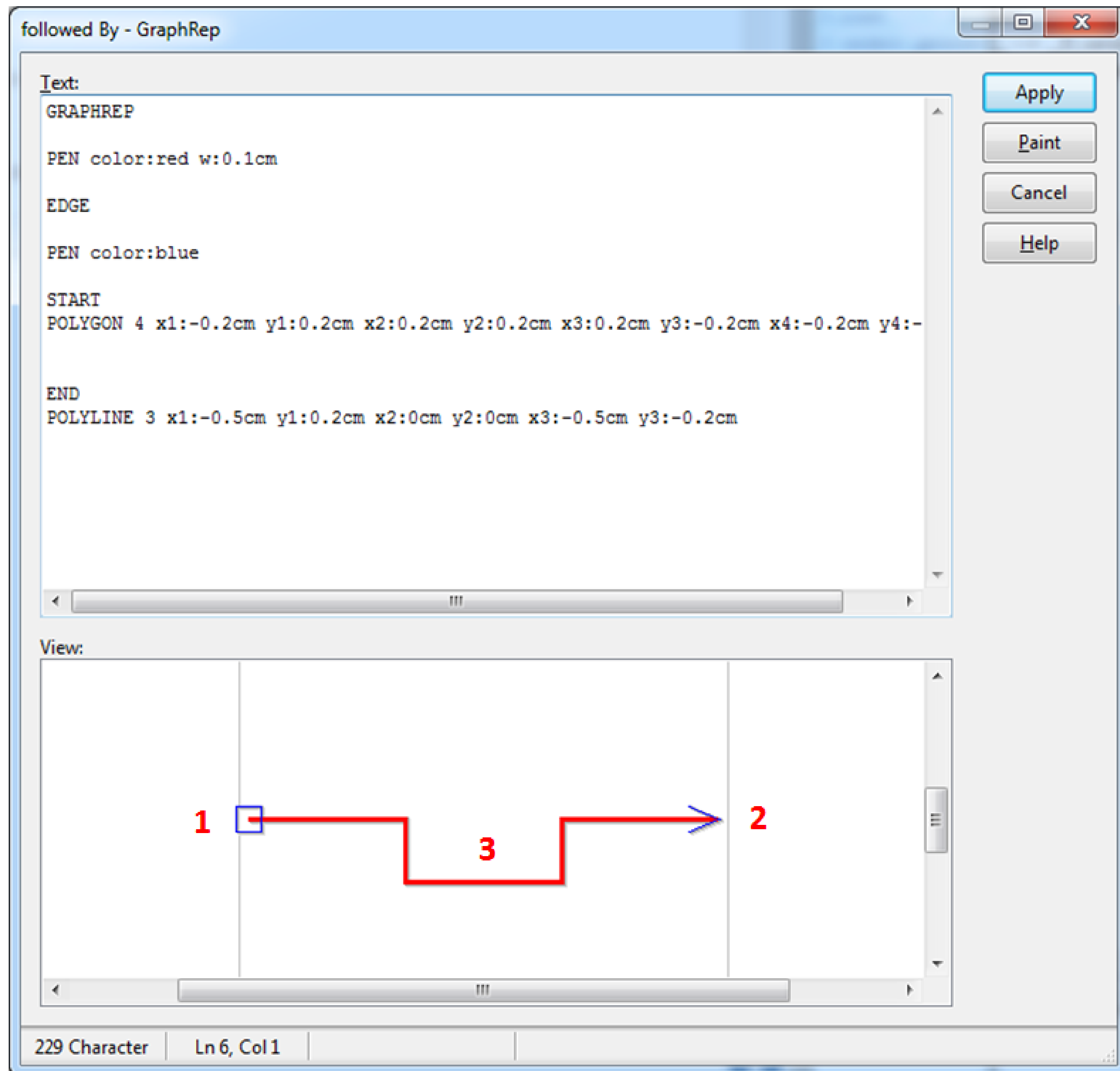


Figura 61 Crearea reprezentării grafice pentru conectori

Avertisment: este nevoie să fiți atenți la modul în care se măsoară coordonatele deoarece nu este intuitiv. Sistemul de coordonate are o orientare diferită pentru START și pentru END după cum se vede în figura de mai jos:

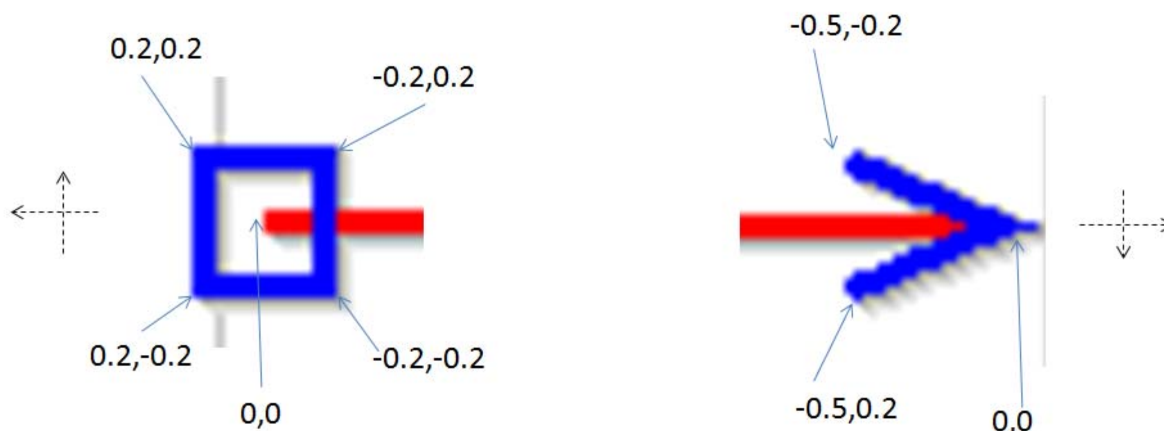


Figura 62 Sistemul de coordonate pentru START și END

Sistemul de coordonate general (ce se aplică și pentru EDGE și pentru GraphRep) este cel din partea dreaptă, cu x orientat spre dreapta și y în jos. Pentru START-ul săgeții sistemul de coordonate este inversat! După ce conceptele și relațiile sunt definite, putem închide fereastra Class hierarchy. Un mesaj ne va avertiza că au fost făcute modificări în "library". Răspundeți pozitiv pentru a salva modificările.

În continuare trebuie să grupăm conceptele noastre pe tipuri de modele. Urmați pașii din figura de mai jos:

1. Selectați acea "library" unde am lucrat până acum
2. Selectați Library attributes
3. Selectați grupul Add-ons
4. Scrieți codul în atributul Modi

```
GENERAL order-of-classes:custom

MODELTYPE "Cooking Recipes"
INCL "CookingStep"
INCL "Start"
INCL "Stop"
INCL "followed By"
```

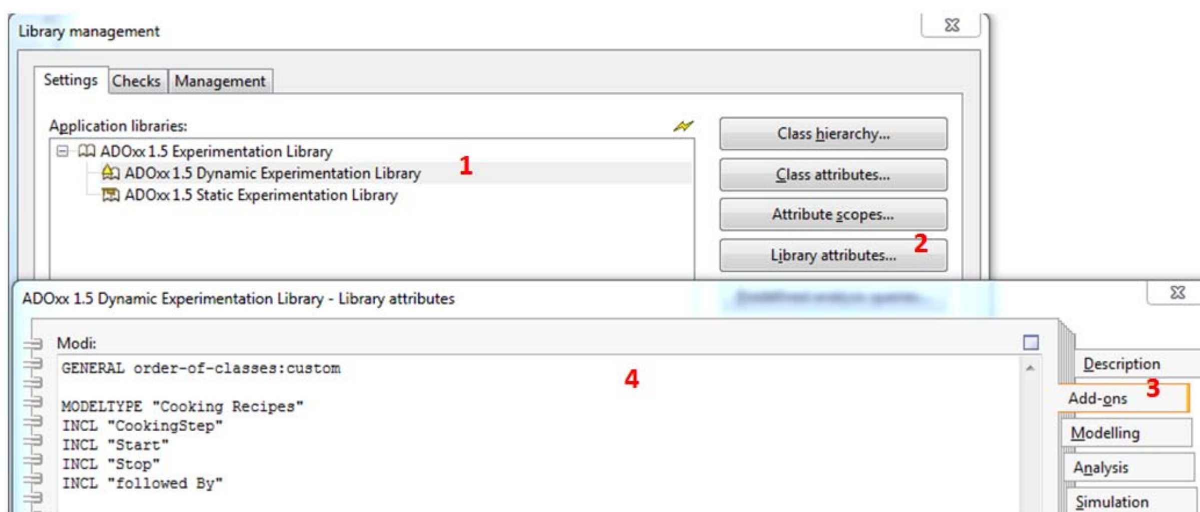


Figura 63 Organizarea conceptelor pe tipuri de modele

Acest exemplu este minimalist:

- comanda GENERAL indică ordinea claselor care vor fi afișate in bară (în instrumentul de modelare) ar trebui să fie cea oferită aici
- comanda MODELTYPE definește numele tipului de model

- comanda INCL include conceptele și relațiile. Observați faptul că Node nu este inclus. Nu are un GraphRep și a fost creat doar ca și concept de unificare (pentru a reprezenta toate tipurile de noduri din diagramă)

Închideți ferestrele confirmând cu Da toate cererile pentru salvarea modificărilor. Este posibil să primiți un mesaj de eroare dacă s-a greșit numele unui concept sau a unei relații!

Ultimul pas înaintea generării instrumentului de modelare este de a atribui un utilizator pentru Experimentation Library. În fereastra User management, urmați pașii de mai jos:

1. Adăugați un nou utilizator (user)
2. Selectați Experimentation Library
3. Alegeți User groups
4. Faceți click pe ADOxx group care include toate drepturile de utilizator (pentru a evita specificarea prea multor detalii)

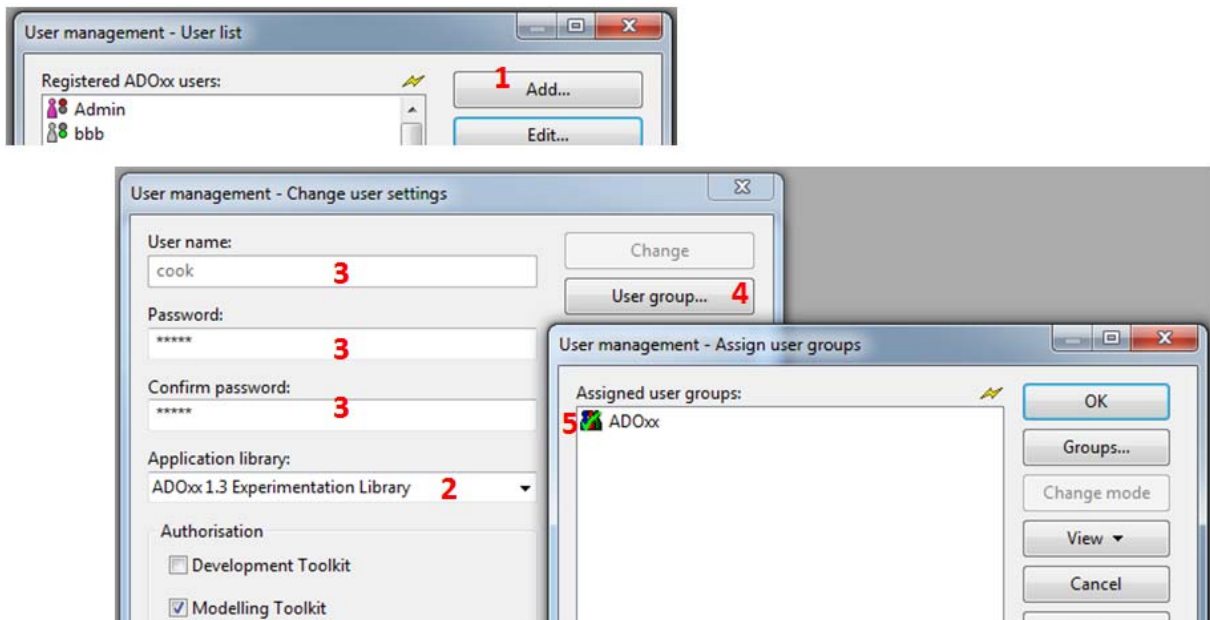


Figura 64 Crearea unui utilizator și parola acestuia pentru metoda de modelare

Acum porniți instrumentul de modelare cu credențialele utilizatorului creat anterior. După logare, observați următoarele:

1. Faceți click dreapta pe folder-ul Models și veți vedea tipul de model (Model Type) pe care l-ați definit (Cooking Recipes)
2. Creați un astfel de model și veți observa bara de instrumente cu concepte, de unde vom putea să selectăm simbolurile acestora. Vă reamintiți că pentru a avea un simbol vizibil pe bara de instrumente a conceptelor, este nevoie să:
    a. definim un GraphRep pentru acesta
    b. să îl includem în tipul de model cu comanda INCL. De aceea nu se vor vedea aici conceptele predefinite din ierarhia conceptelor (precum _D_construct – acestea sunt numite clase abstracte = sunt folositoare pentru moștenirea semanticii, dar nu sunt instanțiate în diagrame)
3. Exersați pe planșa de modelare pentru a vedea cum se poate crea un model. Observați că putem conecta orice de orice și ori de câte ori dorim: se poate crea o săgeată de intrare în Start, o săgeată de ieșire din Stop, mai multe săgeți care să iasă din același Node – asta înseamnă că nu au fost definite restricții sintactice! Singura restricție existentă este cea din ADOxx: nu se poate desena o săgeată de mai multe ori între aceleași perechi de noduri, în aceeași direcție!
4. O altă problemă este că limbajul de modelare nu are nici o semantică. În afară de cuvintele Start și Stop (care sunt parte din notație) nimeni nu poate înțelege ce se urmărește prin această digramă.

Faceți dublu-click pe unul din nodurile CookingStep și veți obține un set aproape gol de proprietăți. Singura proprietate editabilă este Name (moștenită de la _D_construct) și nici măcar aceasta nu este vizibilă pe planșă (pentru a o vedea trebuie deschisă foaia de proprietăți). Asta înseamnă că modelul nostru nu are semantică = un utilizator uman poate să îi dea orice interpretare dorește și nu sunt proprietăți disponibile pentru interogare (altele decât numele). Cu alte cuvinte, modelul nu este altceva decât o diagramă vizuală fără să includă cunoștințe folositoare. Mai mult nivelul vizual (notația) este de asemenea foarte slabă, din moment ce nici măcar numele pașilor de gătit nu este vizibil în mod direct pe diagramă!

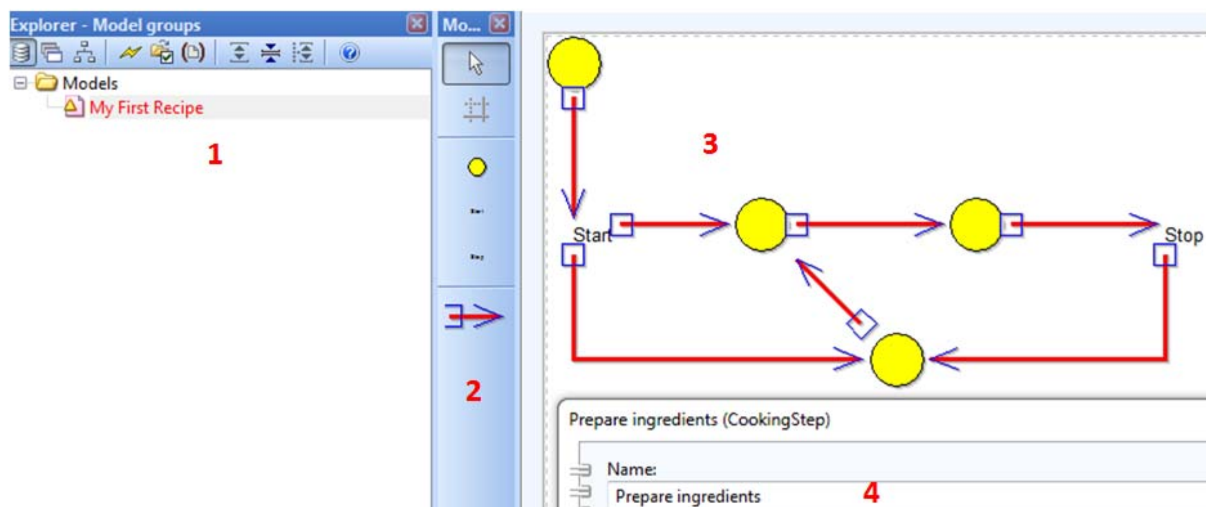


Figura 65 Modelarea unui rețete de gătit în instrumentul de modelare generat

Astfel trebuie să rezolvăm două probleme pentru a crea sintaxă și semantică limbajului nostru:

- să includem niște constrângeri sintactice bine formate
  - nici o săgeată nu ar trebui să lege un Step de un Start, sau punctul Stop de un Step;
  - pașii ar trebui să fie în secvență, astfel nu ar trebui să se permită ca două săgeți să iasă din același Step sau să intre în același Step;
  - un model ar trebui să aibă exact un singur Start și un singur Stop;
- să includem semantică (proprietăți):
  - un Step ar trebui să aibă un slot pentru proprietatea Cost pentru a indica un cost estimativ (de ex. pentru consumul de ingrediente) ca valoare întreagă între 0 și 10000 (doar un interval aleator pentru a demonstra restricția)
  - un conector ar trebui să aibă un slot pentru proprietatea Condition pentru descrie care sunt condițiile necesare pentru a trece la pasul următor, sub forma unui șir de lungime maximă 20 (din nou, o lungime aleatoare în scop demonstrativ)

Ține-ți minte că dacă revenim în aplicația Development Toolkit pentru a face modificările, ar trebui închis instrumentul de modelare iar apoi să ne logăm după ce schimbările în metamodel au fost salvate.

8.2 Elemente de sintaxă

În Development Toolkit, deschideți conceptul CookingStep și definiți Class cardinality cu următorul cod:

```
CARDINALITIES min-outgoing:1 max-outgoing:1 min-incoming:1 max-incoming:1
RELATION "followed By" FROM_CLASS "Stop" max-incoming:0
RELATION "followed By" TO_CLASS "Start" max-outgoing:0
```

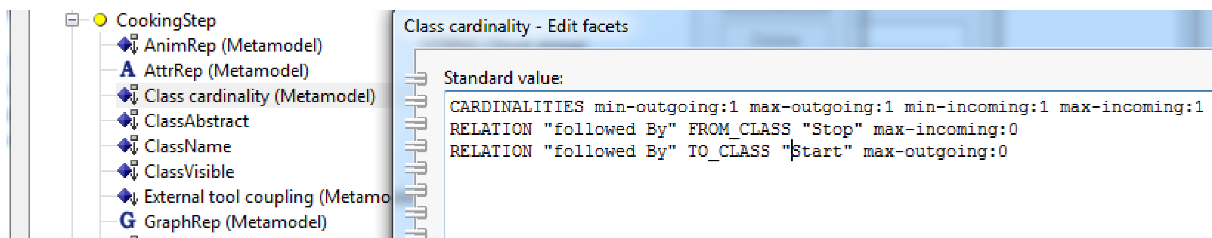


Figura 66 Stabilirea unor restricții de cardinalitate pentru conceptul CookingStep

Acest cod definește următoarele restricții:

- nici un pas (CookingStep) nu ar trebui să rămână neconectat și nici unul nu ar trebui să fie conectat cu mai mult de un conector (atât limita minimă cât și cea maximă sunt puse pe 1 atât pentru săgețile de intrare cât și de ieșire)
- un pas (CookingStep) ar trebui să aibă 0 conectori proveniți dintr-un element Stop
- un pas (CookingStep) ar trebui să aibă 0 conectori care să intre într-un element Start

Salvați și reveniți în instrumentul de modelare. Creați modelul de mai jos și urmăriți numerotarea pentru explicații:

1. Implicit, procesul de modelare este nerestricționat pentru a nu deranja modelatorul cu mesaje de eroare frecvente, în special dacă este posibil să se creeze temporar modele incomplete. În acest caz, verificarea cardinalității este aplicată explicit printr-o opțiune din meniu: Model – Check Cardinalities. Pentru exemplul din imagine, următoarele erori ar trebui să fie detectate:
2. Două săgeți care provin din același pas
3. Săgeată care intră în elementul Start
4. Săgeată care iese din elementul Stop
5. Pas neconectat

Totuși, observați unele erori care încă nu sunt detectate:

6. Este posibil să avem mai mult de un singur element Start și un singur element Stop în cadrul unui model (precum și zero elemente Start sau Stop)
7. Este posibil să avem un conector către elementul Start chiar dacă acesta nu este desenat dintr-un pas CookingStep (și de asemenea să facem un conector dinspre elementul Stop chiar dacă nu va fi dus către CookingStep)

Motivul pentru care sunt permise ultimele erori se datorează faptului că restricțiile ar trebui puse în cadrul conceptelor Start și stop (constrângerile noastre au fost definite doar pentru CookingStep).

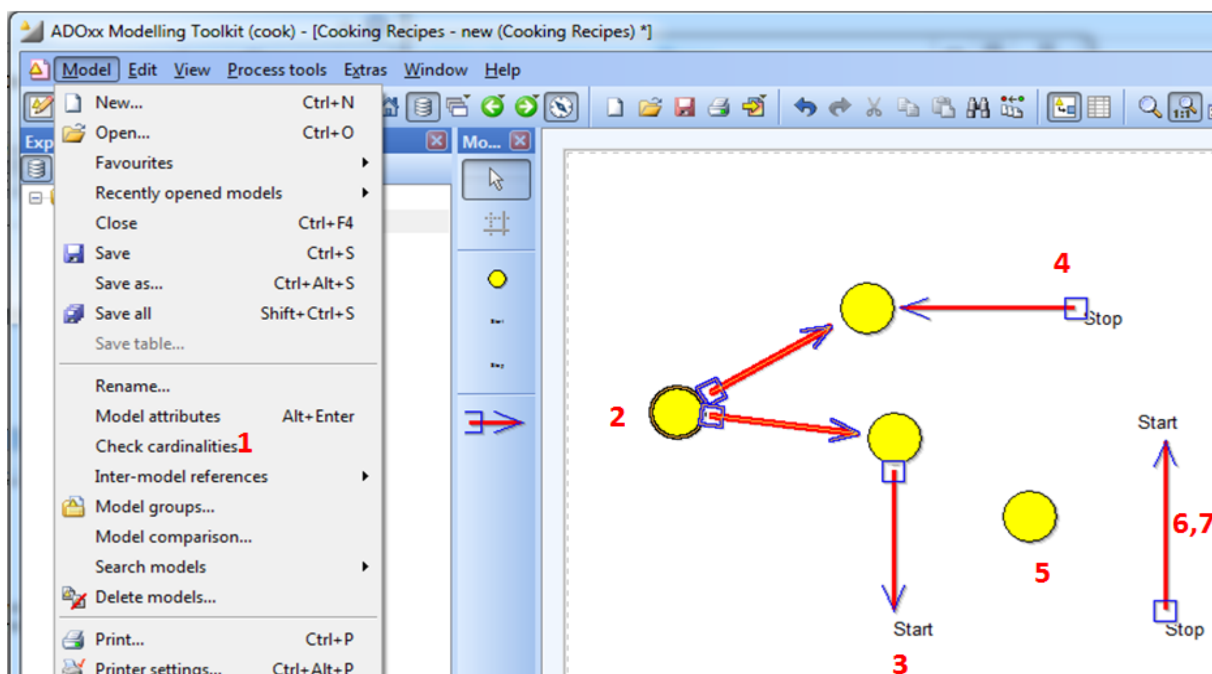


Figura 67 Posibile erori în modelare neacoperite de restricțiile impuse doar asupra conceptului CookingStep

Reveniți în Development Toolkit pentru a schimba acest lucru. Redefiniți toate cardinalitățile în felul următor:

- pentru CookingStep reduceți-le după cum urmează (vom muta toate constrângerile pentru Start și Stop):

CARDINALITIES min-outgoing:1 max-outgoing:1 min-incoming:1 max-incoming:1

- Pentru Start:

CARDINALITIES max-objects:1 min-objects:1
RELATION "followed By" max-incoming:0 max-outgoing:1

- Pentru Stop:

CARDINALITIES max-objects:1 min-objects:1
RELATION "followed By" max-incoming:1 max-outgoing:0

În plus, vom schimba momentul în care se verifică cardinalitățile. Pe lângă opțiunea din meniu menționată anterior, ne vom asigura că restricțiile se vor verifica:

- la evenimente generate de mouse (de ex. la momentul când se desenează un conector se va verifica dacă se încalcă restricțiile)
- la salvare (de ex. pentru a surprinde orice încălcări care nu au fost detectate de evenimente ale mouse-ului precum absența unui anume tip de element).

Acestea se vor defini ca atribute din library:

1. Selectați library
2. Deschideți Library attributes
3. Selectați secțiunea Modelling
4. Editați valorile implicite ale atributului prin adăugarea următoarei linii care va activa verificarea cardinalității atât la evenimentele generate de mouse cât și la operațiile de salvare:

CHECK_CARDINALITIES before-save:1 after-modeling-action:1

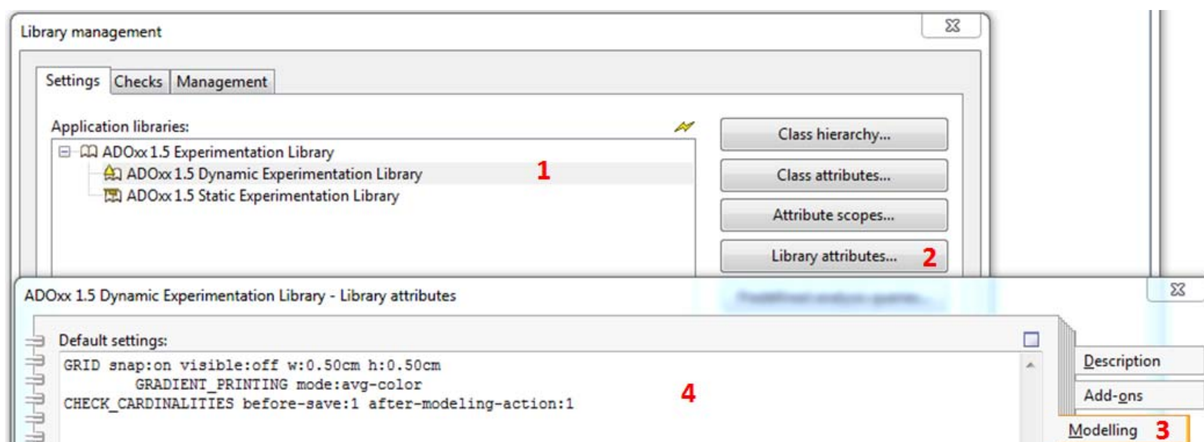


Figura 68 Stabilirea momentului de verificare a restricțiilor de cardinalitate

Salvați și reveniți în instrumentul de modelare. Creați un nou model și încercați să îl salvați fără să adăugați ceva în el. Veți vedea erori precum cea care specifică faptul că cel puțin un element Start ar trebui să fie prezent.

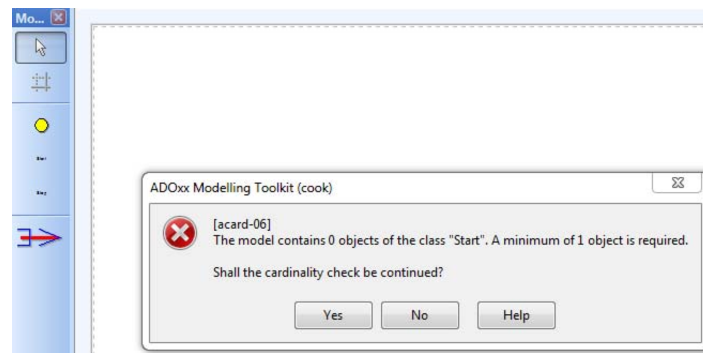


Figura 69 Verificarea restricțiilor de cardinalitate generate de evenimentul de salvare

Apoi încercați să trasați o săgeată de la un element Stop la unul Start. Se va realiza o verificare declanșată de un eveniment de mouse.



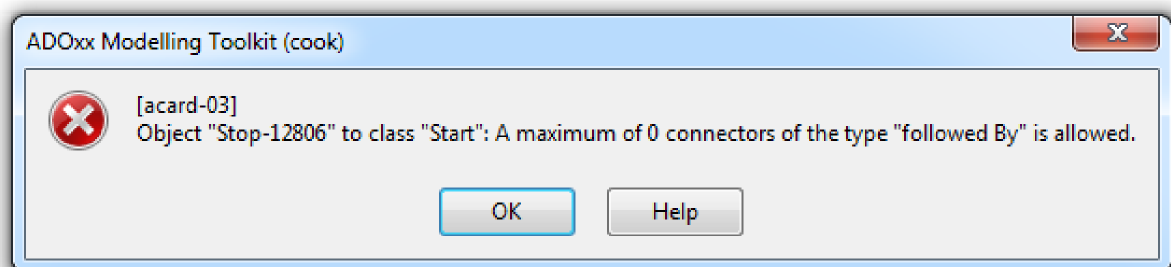


Figura 70 Verificarea restricțiilor de cardinalitate generate de evenimentul de mouse

Toate celelalte constrângeri încă sunt active. Dacă mesajele de eroare sunt deranjante puteți să ștergeți linia cu CHECK_CARDINALITIES din atributele library (probabil veți avea nevoie să salvați anumite modele incomplete pentru unele exerciții).

8.3 Elemente de semantică

În continuare, ne vom ocupa de semantică. În Development Toolkit, alegeți Experimentation Library și îmbunătățim semantica limbajului precum în imaginea de mai jos:

1. faceți click dreapta pe conceptul CookingStep și selectați New attribute
2. să presupunem că dorim să înregistrăm timpul estimativ pentru fiecare pas de gătit. Creați un atribut Cost.
3. selectați tipul acestuia Integer (observați multitudinea de tipuri pentru atribute care le puteți alege – de exemplu ați putea să includeți un atribut Time pentru care există predefinit tipul Time).
4. faceți același lucru și pentru săgeata “followed By” unde vom avea atributul Condition de tip String (pentru a indica într-o formă text care sunt condițiile pentru a trece la pasul următor, de ex. „dacă procesul de fierbere este încheiat”).

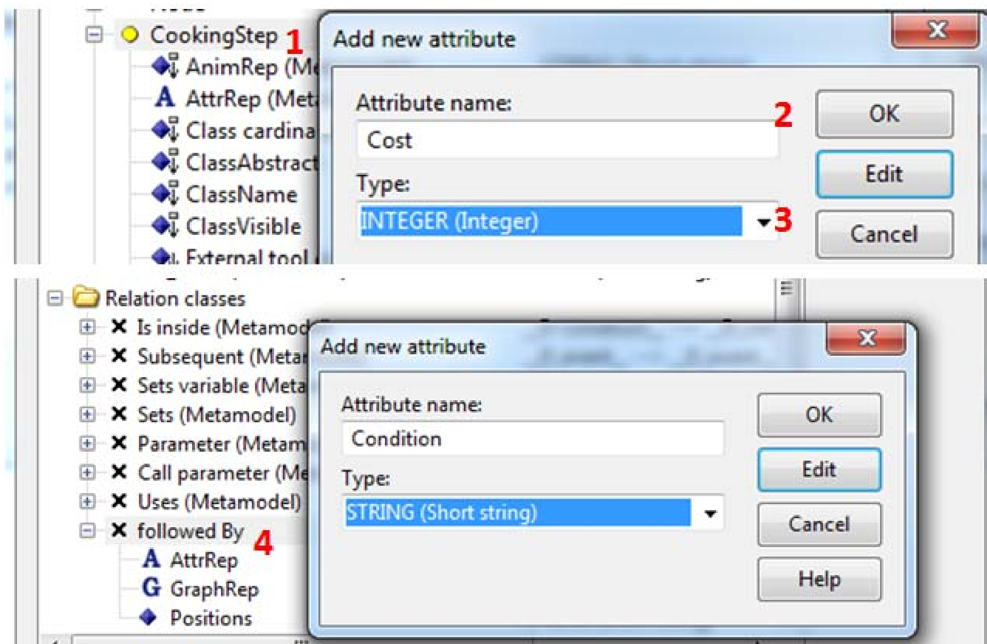


Figura 71 Adăugarea unui nou atribut

Adăugați acum niște restricții suplimentare (intervalul valorilor pentru Cost, lungimea șirului pentru Condition);

1. faceți dublu click pe atributul Cost
2. selectați Facets
3. scrieți următoarele în AttributeNumericDomain (fragmentul de cod stabilește restricția intervalului și un mesaj de eroare):

```
DOMAIN message:"cost should be between 0 and 10000"
INTERVAL lowerbound:0 upperbound:10000
```

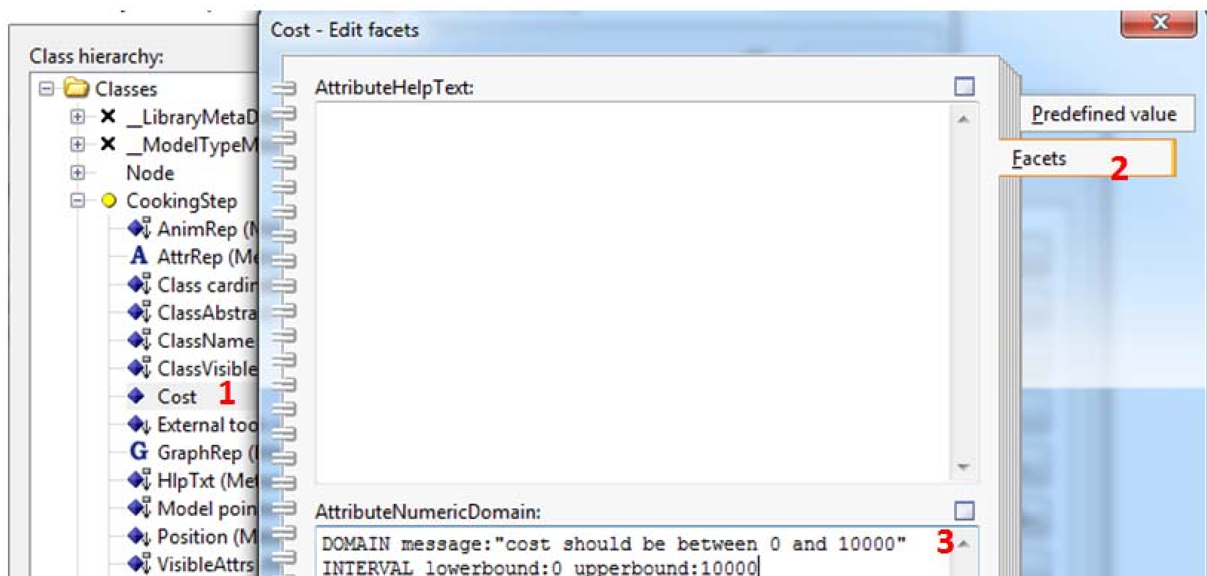


Figura 72 Adăugarea de restricții suplimentare atributului nou creat Cost

Faceți același lucru și pentru lungimea șirului la atributul Condition. Pentru aceasta ne putem folosi de constrângerile date de expresiile regulate:

1. faceți dublu click pe atributul Condition
2. selectați Facets
3. scrieți următoarele în AttributeRegularExpression pentru a impune numărul de caractere să fie între 0 și 20:

```
REGEXP message:"the text should not be longer than 20 characters" expression:"^.{0,20}$"
```

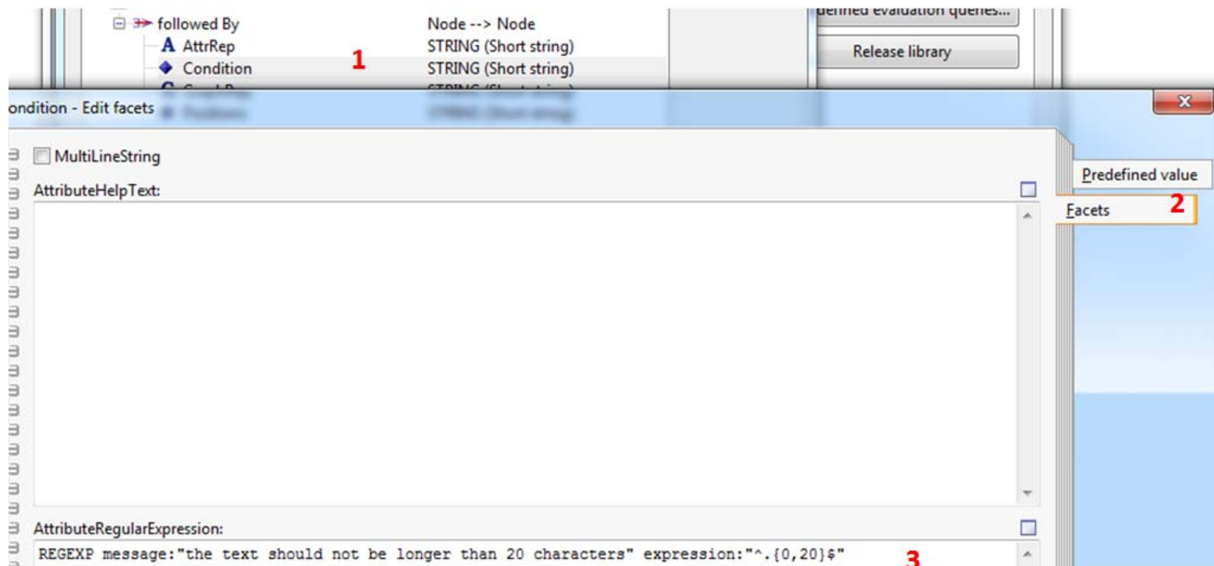


Figura 73 Adăugarea de restricții suplimentare atributului Condition date de expresiile regulate

Acum limbajul este mai bogat în semantică, însă proprietățile trebuie să poată fi editate de către un utilizator. Pentru a face posibil acest lucru, trebuie să indicați în interiorul atributului moștenit AttrRep care sunt proprietățile care ar trebui să fie vizibile de un utilizator:

1. selectați AttrRep pentru CookingStep
2. folosiți sintaxa NOTEBOOK pentru a defini atributele vizibile. Cu cuvântul cheie CHAPTER putem grupa mai multe atribute în interiorul unor pagini (în caz că sunt prea multe atribute). Cu ATTR trebuie să specificați exact numele atributului care va fi făcut vizibil.

NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"
Do the same for the AttrRep of the relation
Make visible the Condition attribute:
NOTEBOOK
CHAPTER "Main attributes"
ATTR "Condition"

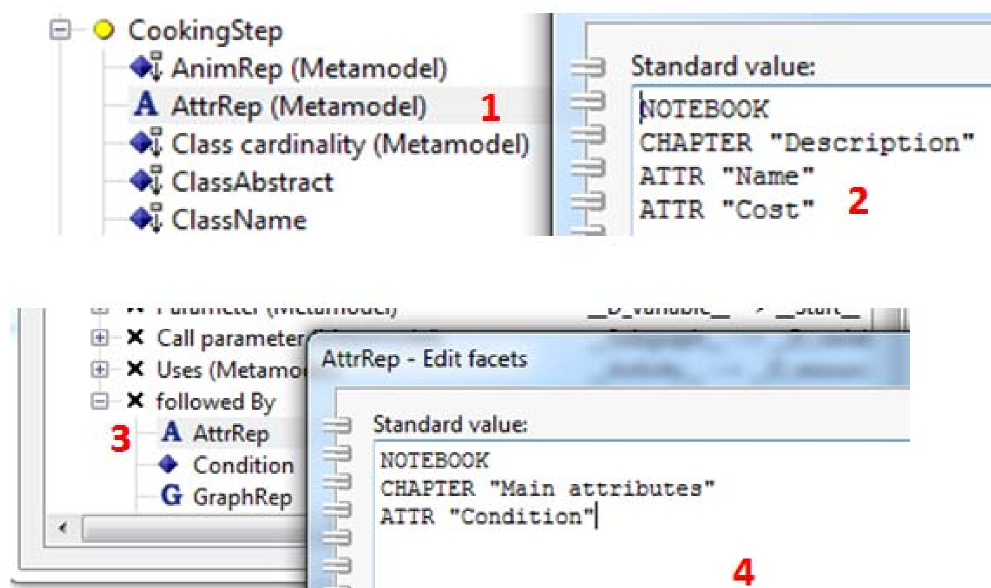


Figura 74 Includerea atributelor nou create în lista de atribute vizibile unui utilizator

Salvați modificările și logați-vă înapoi în instrumentul de modelare. Acum puteți observa:

1. când efectuați dublu click asupra unui pas, puteți să vedeți care sunt proprietățile sale editabile (Name și Cost).
2. când efectuați dublu click asupra unei săgeți, puteți să vedeți Condition
3. mai observați capitolul Main attributes pe care l-am definit pentru a grupa atributele. Mai multe capitole pot fi definite pentru separa atributele pe pagini (când sunt prea multe).

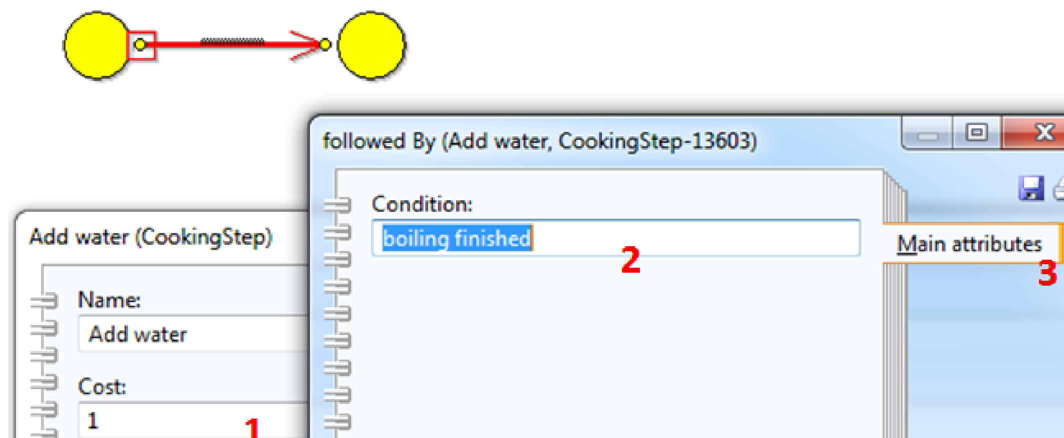


Figura 75 Adăugarea de valori atributelor în instrumentul de modelare

Acum avem un limbaj mai bogat în semantică dar notația încă este destul de rudimentară, nici măcar nu afișează numele elementelor. Trebuie să dezvoltăm notația grafică pentru a comunica din semantica acesteia (valori ale proprietăților).

Reveniți în Development Toolkit și înlocuiți GraphRep de la CookingStep cu următorul cod:[72]

```
GRAPHREP

FONT "Arial" h:12pt color:red
PEN color:red

FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm

ATTR "Name" y:1cm w:c
ATTR "Cost" y:-1cm w:c
```

Observați că am adăugat:

- definiția pentru FONT care va fi folosită de tot textul care îl urmează
- ATTR ce include valorile pentru atributele Name și Cost. Poziția unde sunt afișate acestea, de asemenea este specificată: numele va fi sub centrul simbolului (y:1cm), costul va fi deasupra (-1cm). Parametrul w definește poziția orizontală – pentru a evita stabilirea coordonatelor pentru x, folosim valoarea “c” care va poziționa textul pe orizontală centrat.

Vom face ceva similar pentru relația “followed By” prin includerea atributului Condition în componenta EDGE a săgeții. Deoarece nu specificăm o anumită poziție, va prelua valoarea implicită pentru aceasta (textul va fi afișat începând din mijlocul săgeții).

```
GRAPHREP

PEN color:red w:0.1cm

EDGE
ATTR "Condition"
```

[72] dacă se dorește păstrarea vechiului cod sub formă de comentariu puteți să îl includeți între (* și *)

```
PEN color:blue

START
POLYGON 4 x1:-0.2cm y1:0.2cm x2:0.2cm y2:0.2cm x3:0.2cm y3:-0.2cm  x4:-0.2cm y4:-0.2cm

END
POLYLINE 3 x1:-0.5cm y1:0.2cm x2:0cm y2:0cm x3:-0.5cm y3:-0.2cm
```

Notă: A se consulta ADOxx help pentru sintaxa completă a limbajelor declarative folosite în GRAPHREP, ATTRREP și MODI. Aceste exerciții vor ilustra doar câtiva dintre cei mai frecvenți parametri folosiți.

Logați-vă înapoi în instrumentul de modelare pentru a vedea cum notația a fost extinsă cu semantică (adăugați câteva valori pentru Cost și Condition pentru a afișare).

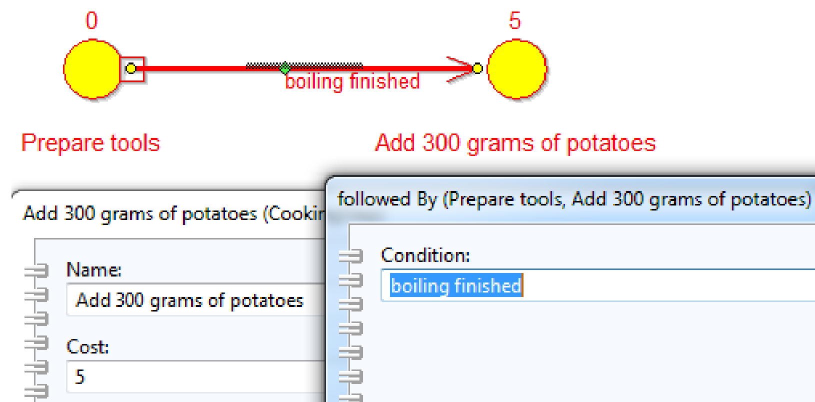


Figura 76 Includerea în notație a valorilor din atribute

8.4 Notație dinamică

Variația notației este realizată când informația de la nivelul vizual se schimbă în mod dinamic odată cu semantica. Cel mai simplu caz de variație este când notația afișează în mod direct valoarea unui atribut. În alte cazuri, este posibil să dorim unele modalități mai subtile de a sugera modificări ale proprietăților. În exemplul următor schimbați GraphRep de la CookingStep:

```
GRAPHREP

AVAL c:"Cost"

FONT "Arial" h:12pt color:red
PEN color:red

FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm

ATTR "Name" y:1cm w:c
TEXT (cond(VAL c>10,"costly step","negligible cost")) y:-1cm w:c
```

Cu AVAL am creat o variabilă c pentru a păstra valoarea din atributul Cost. Variabila ne va permite să programăm în cadrul codului GRAPHREP! (condiții IF, bucle FOR etc.).

În cazul nostru comanda ATTR "COST" a fost înlocuită cu o comandă TEXT. Aceasta, la fel ca la folosirea precedentă, va afișa un text static. Totuși, textul static este stabilit pe baza unui test IF (funcția cond() care verifică dacă valoarea costului este mai mare decât 10 sau nu. Dacă este, textul afișat va fi "costly step". Dacă nu, textul afișat va fi "negligible cost". Testul este realizat asupra VAL c, care convertește un șir către un număr (implicit, valoarea fiecărui atribut este preluată de codul GraphRep ca un șir).

Verificați efectul în instrumentul de modelare, prin stabilirea unor costuri diferite pentru mai mulți pași de gătit:

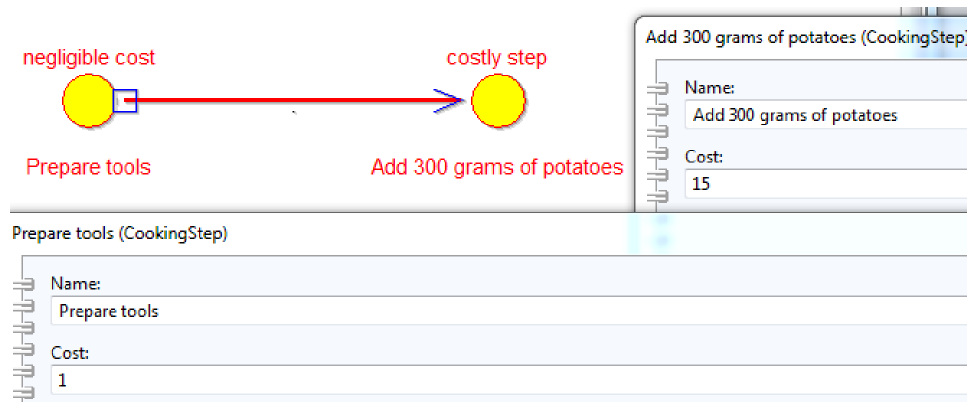


Figura 77 Schimbarea în notație în funcție de valorile atributelor

Variația notației poate fi extinsă pentru a influența și forma grafică. Pentru aceasta, o proprietate numerică a formei grafice (precum dimensiune, poziție, cod culoare, dimensiune font) ar trebui să fie calculată pe baza valorii unui atribut. În următorul exemplu, dimensiunea (raza) cercului galben va fi proporțională cu valoarea atributului cost – 1/5 din valoarea acestuia. De asemenea, trebuie să luăm în considerare ce s-ar întâmpla dacă Costul ar fi zero (un cerc de dimensiune zero ar fi invizibil!). Pentru a evita această situație, vom folosi o structură IF pentru a crea o reprezentare GraphRep implicită pentru situația în care Cost=0.

Înlocuiți GraphRep de la CookingStep cu următorul cod:

```
GRAPHREP
AVAL c:"Cost"
PEN color:red
FILL color:yellow

IF (c="0")
  FONT "Arial" h:10pt color:red line-orientation:45
  ELLIPSE rx:0.2cm ry:0.2cm
  TEXT "cost not estimated" y:-0.5cm
ELSE
  ELLIPSE rx:(CM c/5) ry:(CM c/5)
ENDIF

FONT "Arial" h:14pt color:black
ATTR "Name" y:1cm w:c
```

Observați testul IF care verifică dacă avem zero pentru Cost. În caz afirmativ, un cerc implicit este desenat, cu un text pe diagonală: “cost not estimated”. Dacă nu, valoarea Cost este împărțită la 5 și convertită în CM (centimetri).

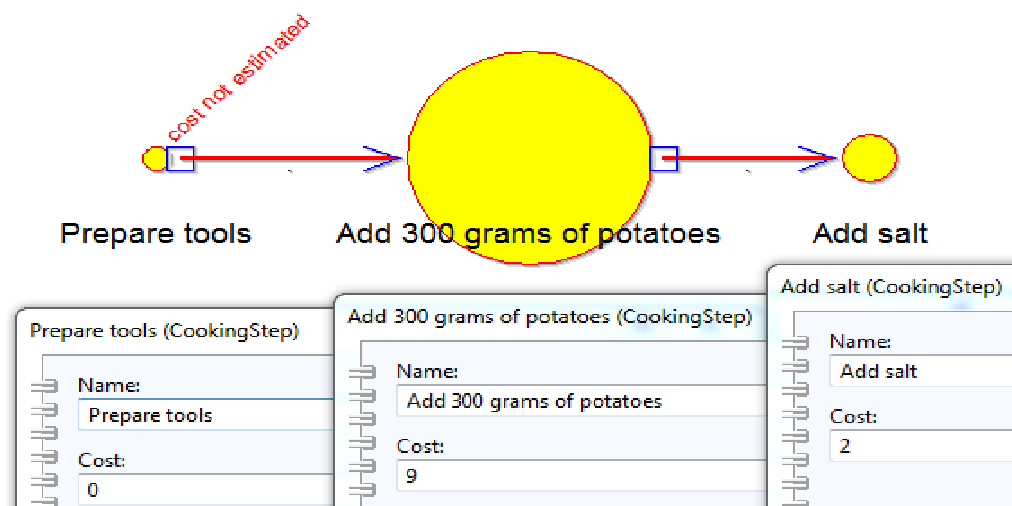


Figura 78 Schimbare reprezentării grafice în funcție de valorile atributelor

Bineînțeles cu o structură IF notația ar putea fi complet schimbată în funcție de anumite condiții puse asupra valorilor din atribute (sau o combinație de condiții). Deci ați putea avea mai multe notații pregătite pentru același concept pentru a comunica diverse stări pentru un element al modelului.

În plus, aveți posibilitatea de a combina forme vectoriale (create cu sintaxa GRAPHREP) și fișiere grafice externe (JPG, PNG, etc.). Dacă doriți să includeți fișiere grafice externe în notația folosită va trebui întâi să le importați în ADOxx în baza de date cu resurse:

1. în meniul Extras, veți găsi opțiunea File Management
2. selectați Experimentation library
3. apăsați Import și alegeți un fișier grafic de pe disk

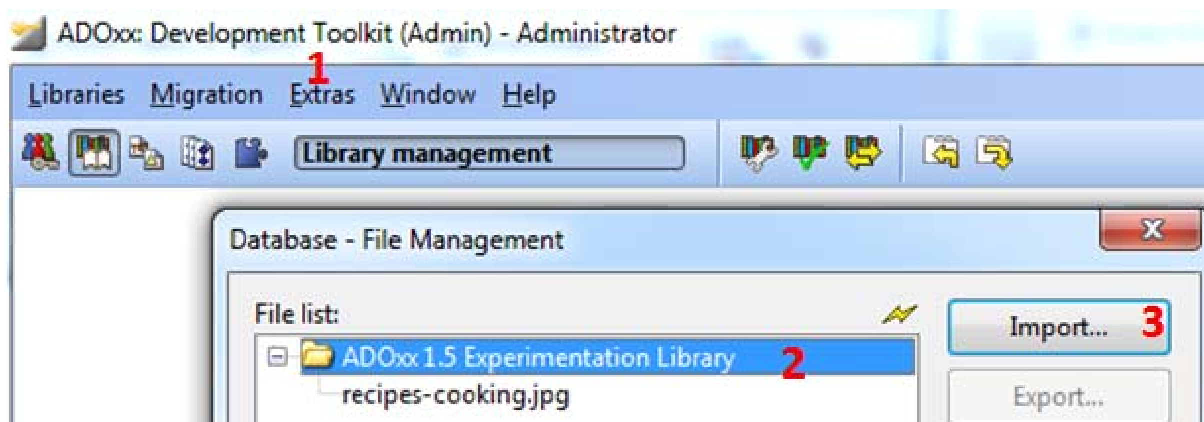


Figura 79 Baza de date cu resurse solicitate de medoda de modelare

Apoi schimbați definiția GraphRep pentru CookingStep prin înlocuirea primei ramuri din IF după cum se vede mai jos:

```
GRAPHREP

AVAL c:"Cost"

PEN color:red
FILL color:yellow

IF (c="0")
  FONT "Arial" h:10pt color:red line-orientation:45
  BITMAP ("db:\\recipes-cooking.jpg") y:-1cm w:1.5cm h:1.5cm
  TEXT "cost not estimated" y:-0.5cm
ELSE
  ELLIPSE rx:(CM c/5) ry:(CM c/5)
```

```
ENDIF

FONT "Arial" h:14pt color:black
ATTR "Name" y:1cm w:c
```

Exemplul presupune că am importat în ADOxx un fișier extern JPG numit „recipes-cooking.jpg”. Calea standard pentru fișierele importate este “db:\\....”.

Comanda BITMAP ne permite să includem în notație fișierul importat și de asemenea să îi controlăm dimensiunea și poziția acestuia. Observați că aceasta ne înlocuiește ELLIPSE care am avut-o anterior, pentru situația în care atributul Cost este zero. Acum în instrumentul de modelare ar trebui să puteți vedea că fișierul JPG înlocuiește cercul pentru nodurile la care nu este specificată nici o valoare pentru Cost. În acest fel puteți include orice număr de fișiere externe grafice pentru a fi afișate în funcție de valori diferite date de semantica conceptului!

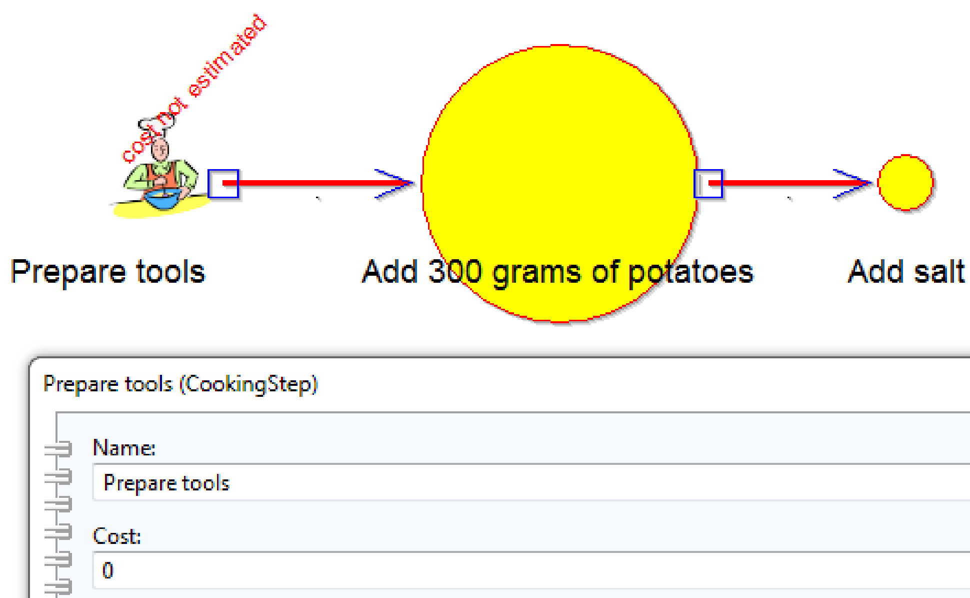


Figura 80 Folosirea unui fișier importat în reprezentarea grafică

Notă: puteți schimba valoarea unui atribut pentru mai multe elemente din model în același timp. Selectați toate elementele cu Ctrl-click și apoi alegeți Edit-Change attributes.

8.5 Extinderea limbajului de modelare

În continuare vom defini un al doilea tip de model, pentru a descrie resursele necesare în timpul procesului de gătit. Conceptul principal aici va fi CookingResource, specializat în două tipuri de resurse: CookingIngredient și CookingObject. Singura relație va fi “always requires” pentru a indica că o anumită resursă întotdeauna solicită un anumit instrument (de ex. laptele solicită mereu un anume recipient – de ex. o sticlă). Vom folosi această relație pentru a descrie dependențele, pentru a conecta orice resursă (atât ingredientele cât și instrumentele/obiectele) de obiecte de care depind (ar trebui folosite împreună).

Creați conceptul CookingResource sub _D_construct (folosiți New class). Creați CookingIngredient și CookingObject sub conceptul CookingResource.
Creați relația “always requires” cu New realtionclass de la orice CookingResource la orice CookingObject.

Definiți următorul GraphRep pentru CookingIngredient (un cerc simplu roșu cu numele sub):

```
GRAPHREP

PEN color:red
```

ELLIPSE rx:1.2cm ry:1.2cm

ATTR "Name"

Definiți următorul GraphRep pentru CookingObject (un dreptunghi simplu)

GRAPHREP

PEN color:blue
RECTANGLE x:-1cm y:-1cm w:2cm h:2cm
ATTR "Name" y:1cm w:c

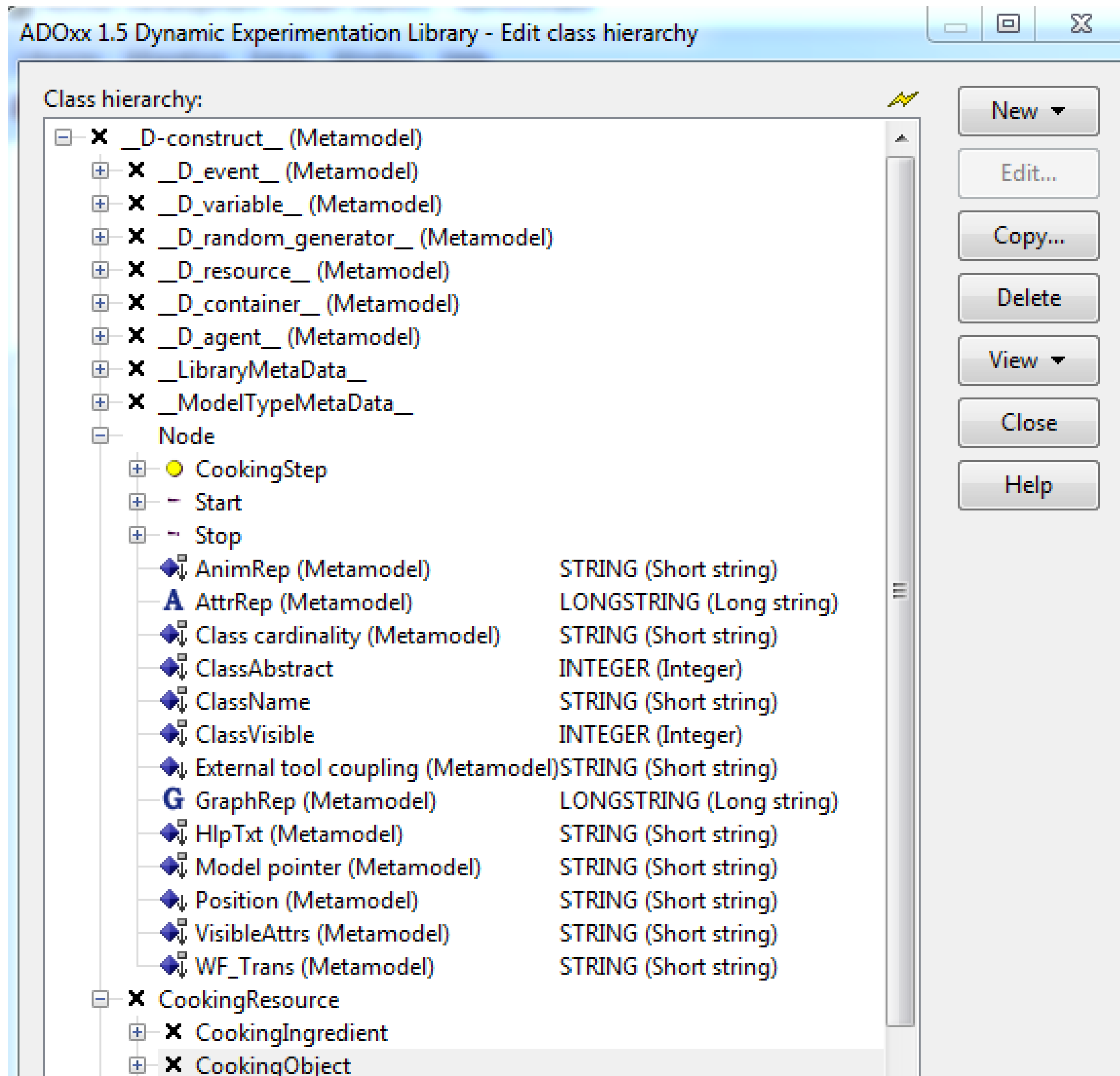


Figura 81 Crearea conceptelor pentru al doilea tip de model cu resursele necesare procesului de gătit

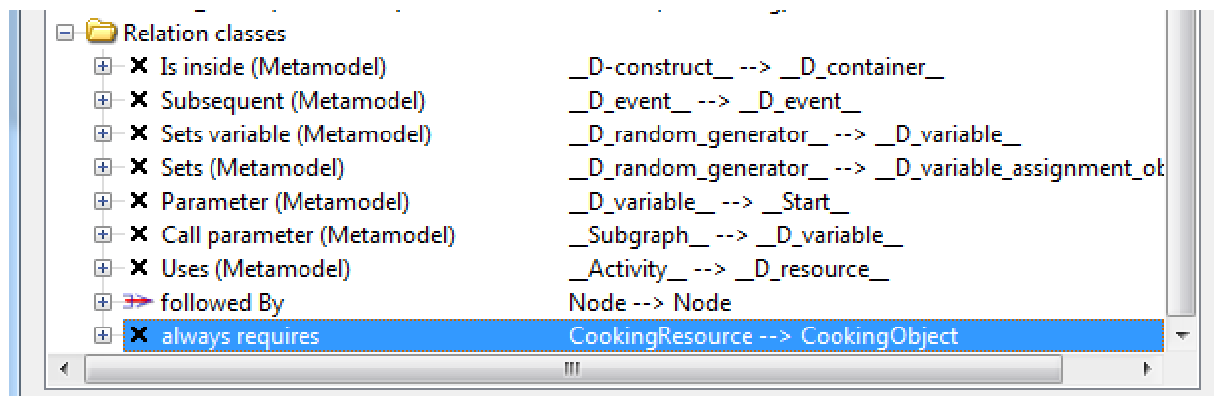


Figura 82 Crearea relațiilor pentru al doilea tip de model

Definiți următorul GraphRep pentru relația "always requires":

```
GRAPHREP

PEN style:dot w:0.1cm endcap:round

EDGE
TEXT "always requires" w:c

END
ELLIPSE rx:0.3cm ry:0.3cm
```

Observați elementele noi:

- PEN are acum un nou stil de linie (punctată) și parametrul endcap care va face punctele ușor rotunjite
- o secțiune START lipsește și din acest motiv nu este definită nici o formă pentru începutul conectorului
- secțiunea EDGE include doar un text static
- secțiunea END va face capul săgeții să pară un cerc.

Acum vom include și un concept container, care este o formă grafică ce poate conține alte simboluri pentru a sugera o anumită grupare. În cazul nostru, va fi numit ComplexIngredient și va fi folosit pentru a grupa mai multe ingrediente. Pentru containere, _D_construct oferă un subconcept dedicat numit _D_containere care la rândul lui oferă două tipuri de containere:

- _D_swimlane ce se comportă ca un obiect swimlane din BPMN, ceea ce înseamnă că se poate întinde pe verticală sau pe orizontală și poate fi mutat sau repoziționat doar pe una din dimensiuni
- _D_aggregation ce este un container mai flexibil – poate fi redimensionat și repoziționat într-un mod mai liber.

Creați ComplexIngredient sub _D_swimlane, pentru a beneficia de comportamentul predefinit (puteți observa și alte specializări pentru _D_construct – sunt cam aceleași pe care le puteți găsi în instrumentul de modelare ADONIS).

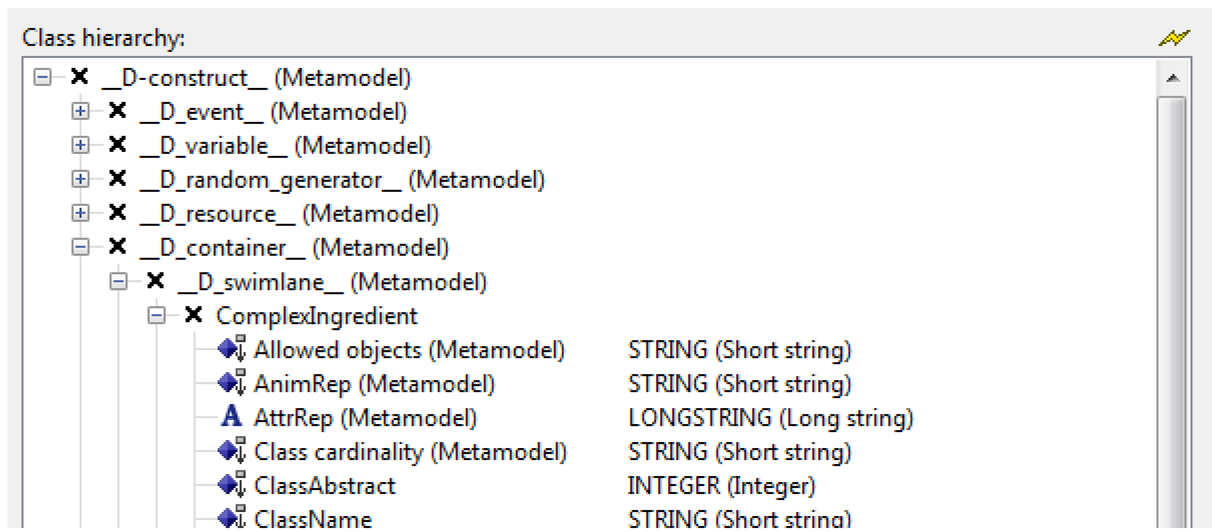


Figura 83 Creare concept ComplexIngredient de tip culoar -swimlane

Următorul fragment de cod este pentru GraphRep:

```
GRAPHREP swimlane:horizontal

PEN w:0.1cm color:green

TABLE w:10cm h:4cm cols:3 rows:1
    w1:3cm w2:1cm w3:100%
    h1:100%
STRETCH off
RECTANGLE x:(tabx0) y:(taby0) w:(tabw1+tabw2+tabw3) h:(tabh1)
LINE x1:(tabx1) y1:(taby0) x2:(tabx1) y2:(taby1)
LINE x1:(tabx2) y1:(taby0) x2:(tabx2) y2:(taby1)

ATTR "Name" x:(tabw1/2) y:(tabh1/2) w:c h:c
```

Explicații:

- Parametrul swimlane stabilește reprezentarea să fie pe orizontală (adică culoarul să se întindă pe întreaga lungime a suprafeței de modelare, redimensionarea și repoziționarea sunt posibile doar pe axa verticală)
- Comanda TABLE definește un cadru de tabel cu 3 coloane și 1 linie. Parametrii w1-w3 definesc lățimea pe 3 coloane (ultima va ocupa întregul spațiu rămas). Parametrul h1 specifică ocuparea întregului spațiu deoarece tabelul are doar 1 rând.
- Comanda ATTR inserează numele în centrul primei celule.

Observați că prin TABLE se definește doar un cadru de tabel invizibil, dar nu se va desena bordura tabelului! TABLE doar generează niște nume speciale pentru coordonatele colțurilor de tabel și dimensiunea celulelor după cum se vede în figura de mai jos:

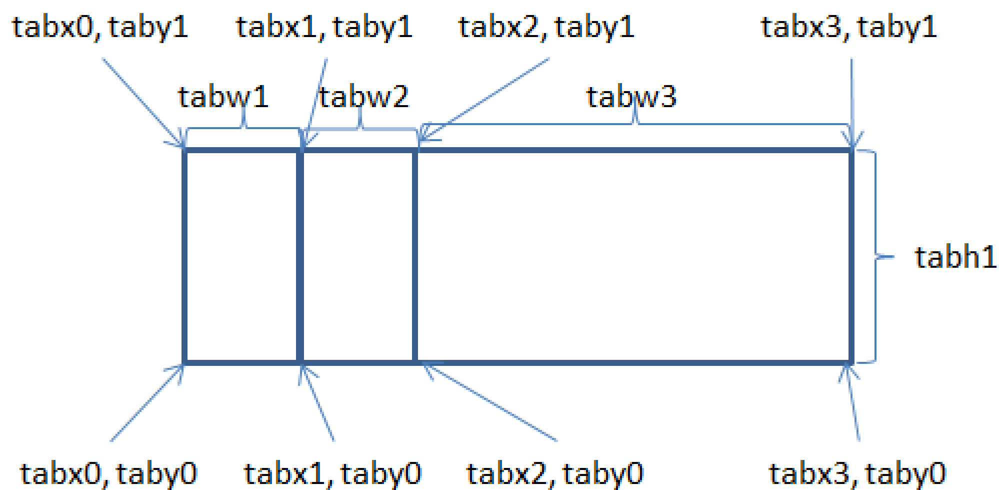


Figura 84 Împărțirea culoarului sub forma unui tabel și coordonatele colțurilor de tabel

Pentru a avea bordurile, trebuie să folosim comenzile de desenare obișnuite (LINE, RECTANGLE, POLYLINE etc.); prin folosirea acestor cuvinte cheie:

RECTANGLE este folosit pentru a desena bordura de exterior. Vom porni din tabx0 și taby0 (nume predefinite pentru primul colț al tabelului), folosim o lățime obținută prin adunarea de la tabw1 la tabw3 (nume predefinite pentru lățimile celor trei coloane) și o înălțime tabh1 (nume predefinit pentru înălțimea primului rând din tabel).

LINE este folosit pentru a desena cele două borduri interioare prin folosirea ca și coordonate numele colțurilor" definite de cadrul tabelului.

În continuare, trebuie să restricționăm ce elemente sunt permise în interiorul culoarului. Deoarece este un ingredient complex, vom permite doar gruparea ingredientelor. Pentru aceasta, scrieți următoarele în atributul predefinit "Allowed objects":

```
ALLOWED from:none INCL "CookingIngredient"
```

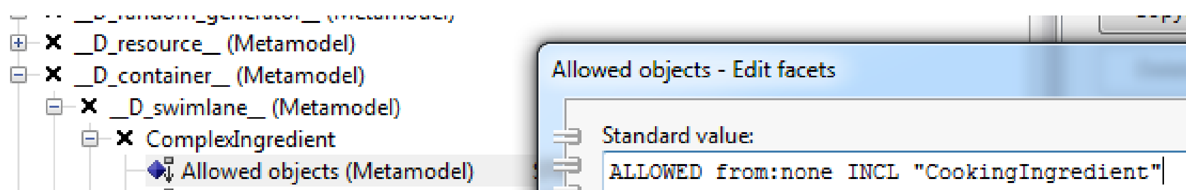


Figura 85 Restricționarea elementelor permise în interiorul culoarului

Să definim acum un tip de model nou care va permite să folosim toate conceptele noi precum și relația nouă. La fel ca mai devreme, alegeți Library Attributes, găsiți atributul Modi unde este definită structura generală a limbajului în termeni de care concepte aparțin cărui tip de model:

```
GENERAL order-of-classes:custom

MODELTYPE "Cooking Recipes"
INCL "CookingStep"
INCL "Start"
INCL "Stop"
INCL "followed By"

MODELTYPE "Cooking Resources"
INCL "ComplexIngredient"
INCL "CookingIngredient"
INCL "CookingObject"
```

```
INCL "always requires"
MODE "Cooking with complex ingreds" from:all
MODE "Cooking without complex ingreds" from:all
EXCL "ComplexIngredient"
```

Observați că am adăugat un tip de model "Cooking Resources" care include noile concepte și noua relație. În plus, am definit și două moduri (modes):

- unul care ne permite să folosim toate aceste concepte (from:all)
- unul care oferă un set redus de concepte (from :all cu excepție ComplexIngredient).

Modurile sunt folosite când numărul de concepte într-un tip de model este foarte mare și nu ar încape toate în bara cu conceptele de modelare. În acest caz, tipul modelului ar putea avea niște versiuni simplificate în care nu ar fi disponibile toate conceptele. În cazul nostru avem posibilitatea să ignorăm acel culoar (swimlane).

Deschideți instrumentul de modelare. Faceți click dreapta pe bara de concepte de modelare care ne va permite să alegem modul dorit. În cel de-al doilea mod, containerul pentru ingrediente complexe nu este disponibil.

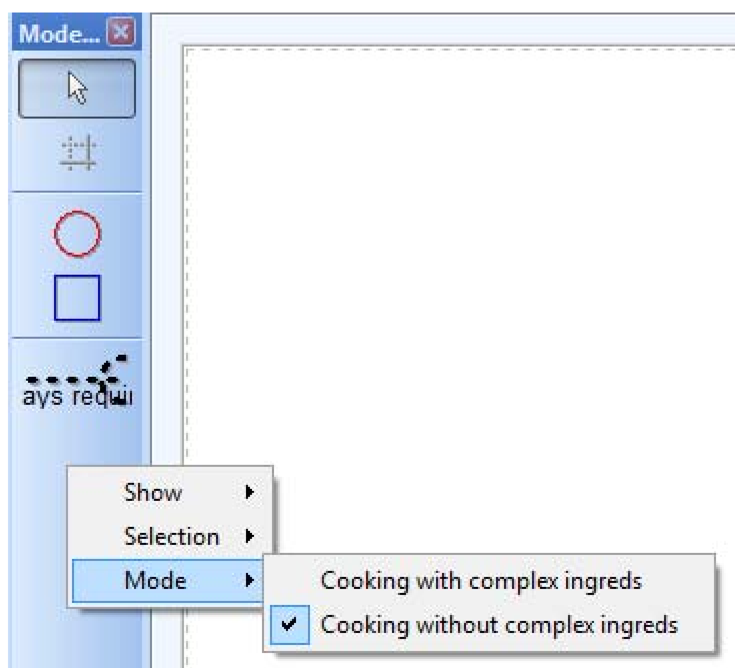


Figura 86 Modurile de lucru posibile în tipul de model Cooking Resources

Selectați modul complet ("Cooking with complex ingreds"), care ne permite să creăm un model precum cel din figura următoare

- observați că CookingObjects nu pot fi introduse în swimlane
- observați că swimlane poate fi redimensionat doar pe verticală. Dacă se adaugă mai multe swimlane pot fi mutate sus și jos pentru a le reordona
- observați că conectorul punctat nu poate lega două ingrediente, dar poate conecta un ingredient de un obiect de gătit sau două obiecte de gătit

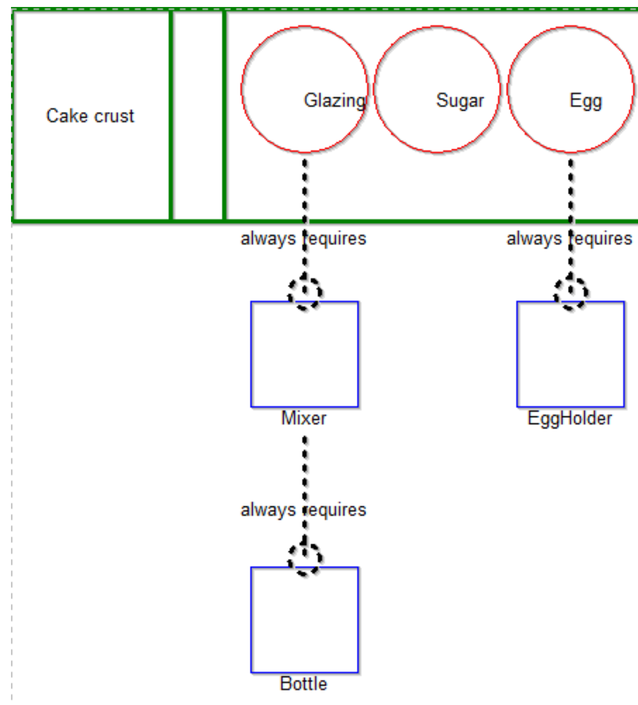


Figura 87 Modelarea resurselor și obiectelor de gătit

Mai mult, remarcați o funcționalitate interesantă care este implicită, bazată pe definiția sintactică a limbajului nostru:

1. butonul evidențiat cu (1) activează un asistent de modelare “modelling assistant”
2. de fiecare dată când selectați un element, veți putea vedea în apropierea acestuia formele conceptelor care sunt permise a fi legate de acesta, în funcție de definiția din metamodel (aici, cercul și pătratul). Selectând unul din acestea se va genera în mod automat un element de tipul respectiv plus conectorul către acesta (de aceea e un mecanism de utilizare definit deasupra sintaxei limbajului!)

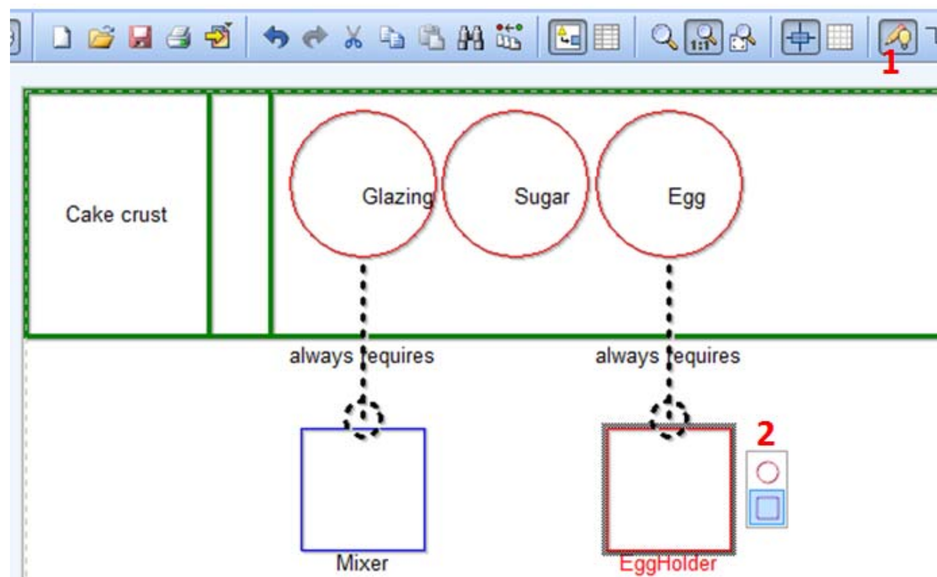


Figura 88 Asistență în procesul de modelare

8.6 Interogarea modelelor

O altă funcționalitate predefinită care se adaptează după sintaxa și semantica limbajului este motorul de interogări care ne permite să executăm interogări asupra conținutului din model, indiferent de tipul de model:

1. În primul rând, selectați butonul marcat cu (1) pentru a activa meniul de analiză (Analysis)
2. Apoi selectați din meniul Analysis - Queries/Reports urmat de alegerea modelului pe care doriți să îl interogați
3. Mai multe tipuri de interogări sunt disponibile – unele dintre ele se bazează pe semantică (să se obțină toate obiectele dintr-o anumită clasă, să se obțină toate obiectele cu o anumită valoare pentru un anume atribut), altele pe sintaxă (să se obțină toate obiectele conectate de un anume obiect sau de un anume conector etc.). Diverse liste verticale ne permit să alegem conceptele, relațiile și proprietățile care sunt relevante pentru modelul pe care dorim să îl interogăm. În exemplul curent vom crea o interogare care va extrage toate obiectele conținute în culoarul Cake crust. Remarcați că am selectat relația "is inside" care e o relație predefinită pe care ADOxx o generează între orice container și elementele conținute (o relație bazată pe poziție și nu pe un conector vizibil!).
4. Partea de jos afișează o interogare în limbajul AQL (limbajul de interogare intern de la ADOxx)
5. Butonul Execute pentru a executa interogarea
6. Zona (6) va afișa rezultatele

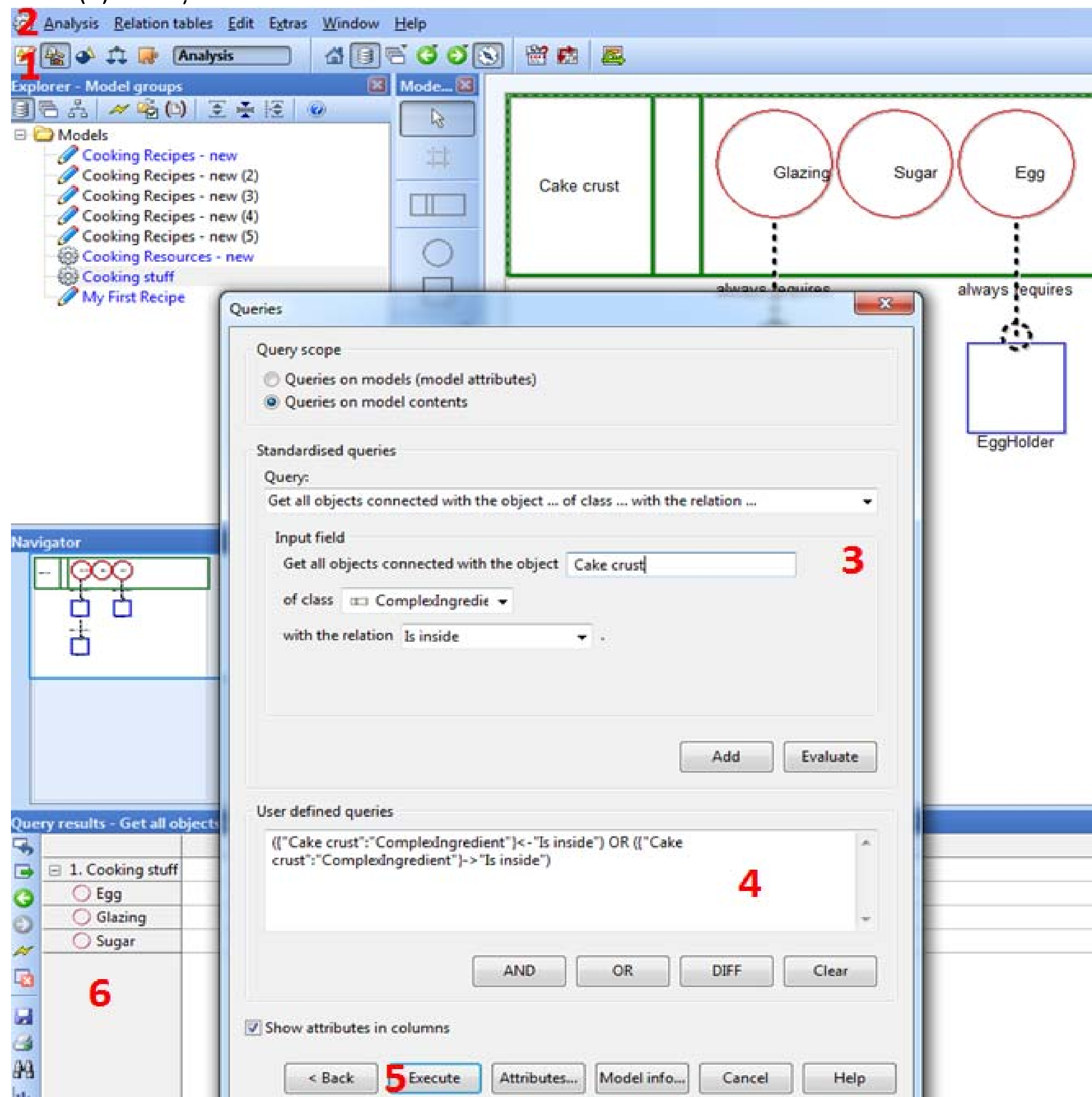


Figura 89 Execuția de interogări asupra conținutului din modele

Încercați să folosiți același motor de interogare în cadrul unui model Cooking Recipe și veți observa că lista de selecție se va adapta după conceptele prezente în acel model. Următorul exemplu construiește o interogare bazată pe semantică: "să se obțină toți pașii de gătit cu un cost < 5"

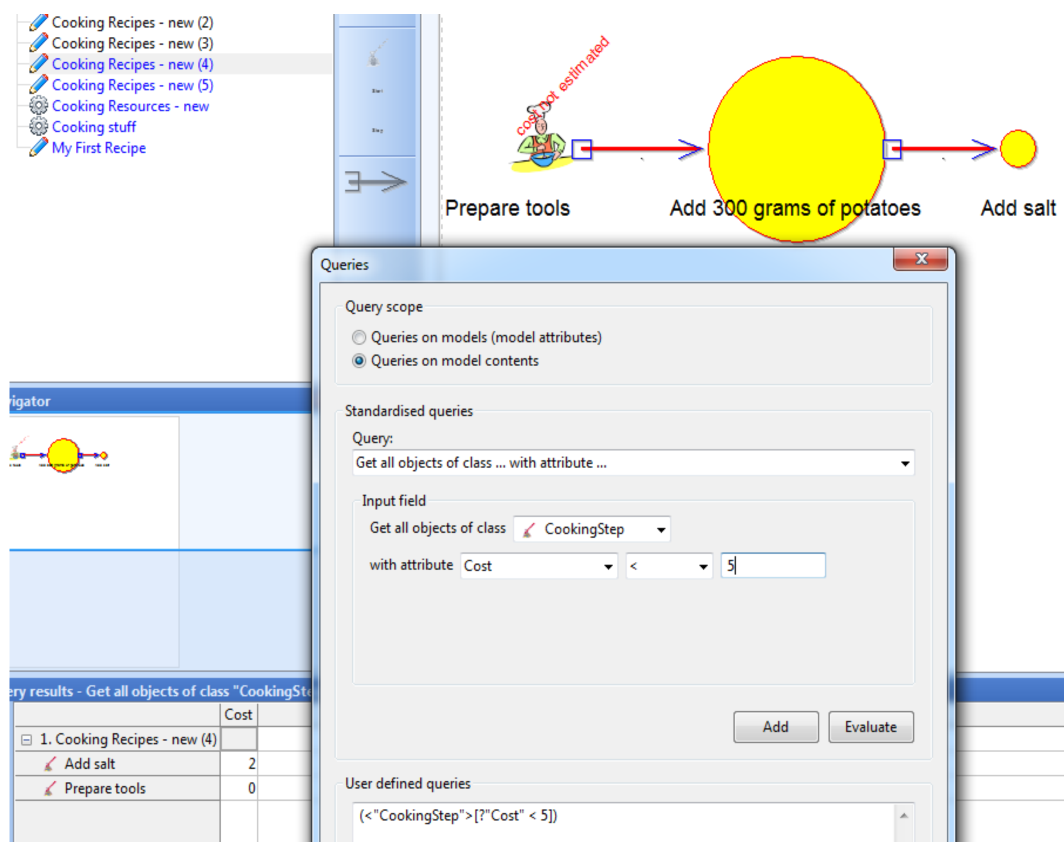


Figura 90 Execuția de interogări asupra atributelor corespunzătoare obiectelor din model

Pe măsură ce adăugăm semantică fiecărui tip de model, posibilitățile de interogare vor fi îmbogățite. Totuși, AQL este limitat în sensul că nu poate accesa date ce sunt externe instrumentului de modelare și nu poate crea interogări ce sunt între modele multiple. De aceea este nevoie să exportăm cunoștințele capturate în diagrame în sintaxă RDF – un format semantic care va păstra relațiile dintre modele și ne va permite să conectăm modele cu orice date externe.

8.7 Atribute specializate

În continuare, vom extinde semantica modelelor cu câteva proprietăți suplimentare:

- CookingObject va avea o proprietate "Type" pentru a indica că este un instrument, recipient sau echipament (Tool, Recipient, Equipment). Bineînțeles am putea crea noi concepte pentru acestea, adăugând unele subclase în ierarhia de concepte. Dar acest lucru este necesar doar în următoarele cazuri:
    - o dacă trebuie să se aplice o anume specializare semantică (de ex. proprietăți distincte care să fie adăugate fiecărui concept specializat). Dacă tot ceea ce dorim este să avem o distincție în ceea ce privește tipul, atunci ar fi suficientă o listă cu selecții. Chiar dacă dorim să avem și o distincție grafică, putem programa GraphRep pentru a arăta diferit în funcție de valoarea atributului type (după cum s-a arătat deja)
    - o dacă dorim să oferim acces la click pentru conceptele specializate: modelatorul va putea să le preia cu un singur click din bara de concepte. Dacă vom adăuga tipul ca proprietate, atunci modelatorul trebuie să deschidă foaia de proprietăți (notebook) pentru a stabili tipul. Aceasta este o problemă de utilizare și este influențată de cât de multe concepte sunt disponibile pe bara de concepte (dacă sunt prea multe, modelatorul ar putea să le reducă folosind proprietatea type!)
- CookingObject de asemenea va primi un atribut pentru descrierea folosinței ("Equipment usage") care va deveni activ doar dacă opțiunea Equipment este selectată din lista cu tipuri.

- Modelele pot avea atribute la nivel de model (“model-level attributes”). Acestea sunt proprietăți care pot fi editate prin selectarea Model Attributes cu click dreapta în zona liberă de pe planșa de modelare. Acestea sunt proprietăți ale modelului ca întreg, numite și “metadate”. Implicit orice model are proprietăți precum Author, Creation date, Keywords, Description, sau proprietăți precum dacă este modelul unei situații planificate (”To be”) sau modelul unei situații existente (”As-is”) etc. Acestea pot fi înlocuite cu proprietățile noastre proprii pentru model. Vom adăuga următoarele atribute pentru modelul CookingRecipe:
    - ”contributor list” un șir pe mai multe linii în care modelatorul poate să scrie numele persoanelor care au contribuit cu adăugarea cunoștințelor în model
    - ”appropriate for” va fi o listă de opțiuni ce permit multiselecție pentru a indica dacă mâncarea descrisă în model este adecvată pentru Micul Dejun, Prânz, Cină sau o combinație a acestora

Pentru început creați o proprietate nouă Type pentru CookingObject. Faceți click dreapta pe concept selectați New attribute. Dați numele Type și pentru tipul acesteia alegeți ”Enumeration”. Aceasta ne va permite să avem o listă de selecție cu valori unice care poate fi afișată în mai multe modalități: drop down, radio buttons, check boxes. După crearea atributului, se va deschide fereastra de mai jos:

1. definiți o valoare care va fi inclusă în listă
2. apăsați Add, și apoi repetați pasul anterior pentru toate valorile care sunt prezentate modelatorului
3. potriviți ordinea valorilor în mod convenabil
4. apăsați Apply

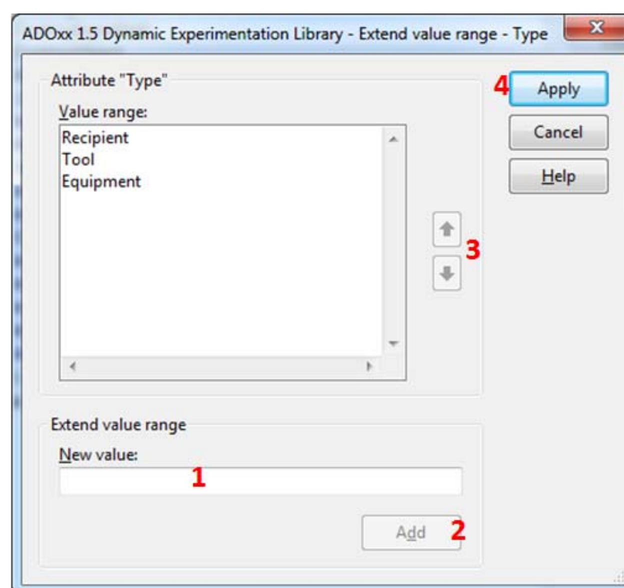


Figura 91 Enumerarea valorilor permise pentru atributului ”Type”

După crearea atributului, faceți dublu click pe acesta pentru a selecta o valoare implicită.

Apoi, creați atributul ”Equipment usage” de tip SRING. În zona Facets a atributului, bifați opțiunea MultilineString pentru a putea scrie descrieri text mai lungi.

Vom face apoi ambele atribute vizibile în foaia de proprietăți, folosindu-ne de atributul AttrRep de la CookingObject.

- pentru atributul Type, indicați felul în care va fi afișată lista de selecție (radio, dropdown etc.), prin folosirea parametrului ctrltype
- pentru atributul ”Equipment usage”, indicați că ar trebui să fie activ doar dacă se selectează valoarea ”Equipment” în atributul Type. Pentru aceasta vom prelua valoarea de la Type cu AVAL la fel cum am procedat mai devreme în GraphRep. Parametrul enabled ne permite să declarăm când

slotul ”Equipment usage” ar trebui să devină activ (în funcție de un test boolean aplicat asupra valorii Type).

```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Type" ctrltype:dropdown
AVAL t:"Type"
ATTR "Equipment usage" enabled:(t="Equipment")
```

În instrumentul de modelare ar trebui să observați acest comportament în foaia de proprietăți pentru CookingObject: caseta pentru ”Equipment usage” ar trebui să devină activă doar dacă tipul ”Equipment” este selectat la Type:

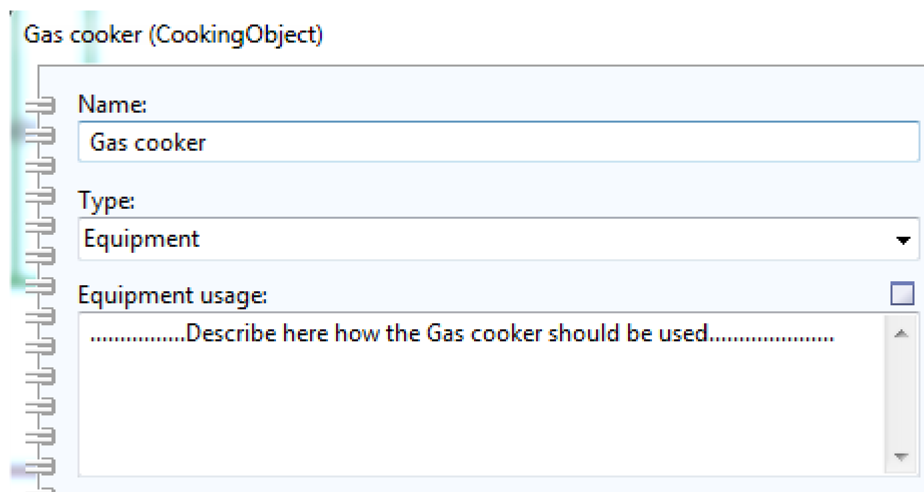


Figura 92 Atributul ”Equipment usage” este activ doar pentru valoarea ”Equipment” din lista ”Type”

Reveniți în Development toolkit:
Crearea atributelor la nivel de model este puțin mai complicată. Acestea sunt realizate în următorii pași:

- toate atributele la nivel de model trebuie să fie definite în clasa predefinită __ModelTypeMetaData__, indiferent de tipul de model în care vor fi incluse
- un alt atribut trebuie creat în aceeași clasă, de tip STRING și ca și conținut codul AttrRep folosit pentru face vizibile atributele
- în Library Attributes, unde sunt definite tipurile de modele, acestea pot primi un parametru attrep a cărui valoare trebuie să fie numele unui atribut STRING unde a fost salvat codul AttrRep

Pașii pot fi urmăriți în următoarea figură:

1. localizați clasa predefinită __modelTypeMetaData__
2. creați un atribut numit ”appropriate for” de tip ENUMERATIONLIST. Acesta ne va permite să creăm o listă multiselecție.
3. includeți opțiunile pentru selecție: Breakfast, Lunch, Dinner
4. creați atributul ”contributor list” de tip STRING
5. bifați opțiunea MultiLineString pentru a avea linii multiple (pentru a scrie pe mai multe rânduri)
6. creați atributul ”CookingRecipe model attributes” de tip STRING
7. scrieți codul AttrRep pentru a face vizibile atributele create anterior:

```
NOTEBOOK
CHAPTER "New model-level attributes"
ATTR "contributor list"
ATTR "appropriate for"
```

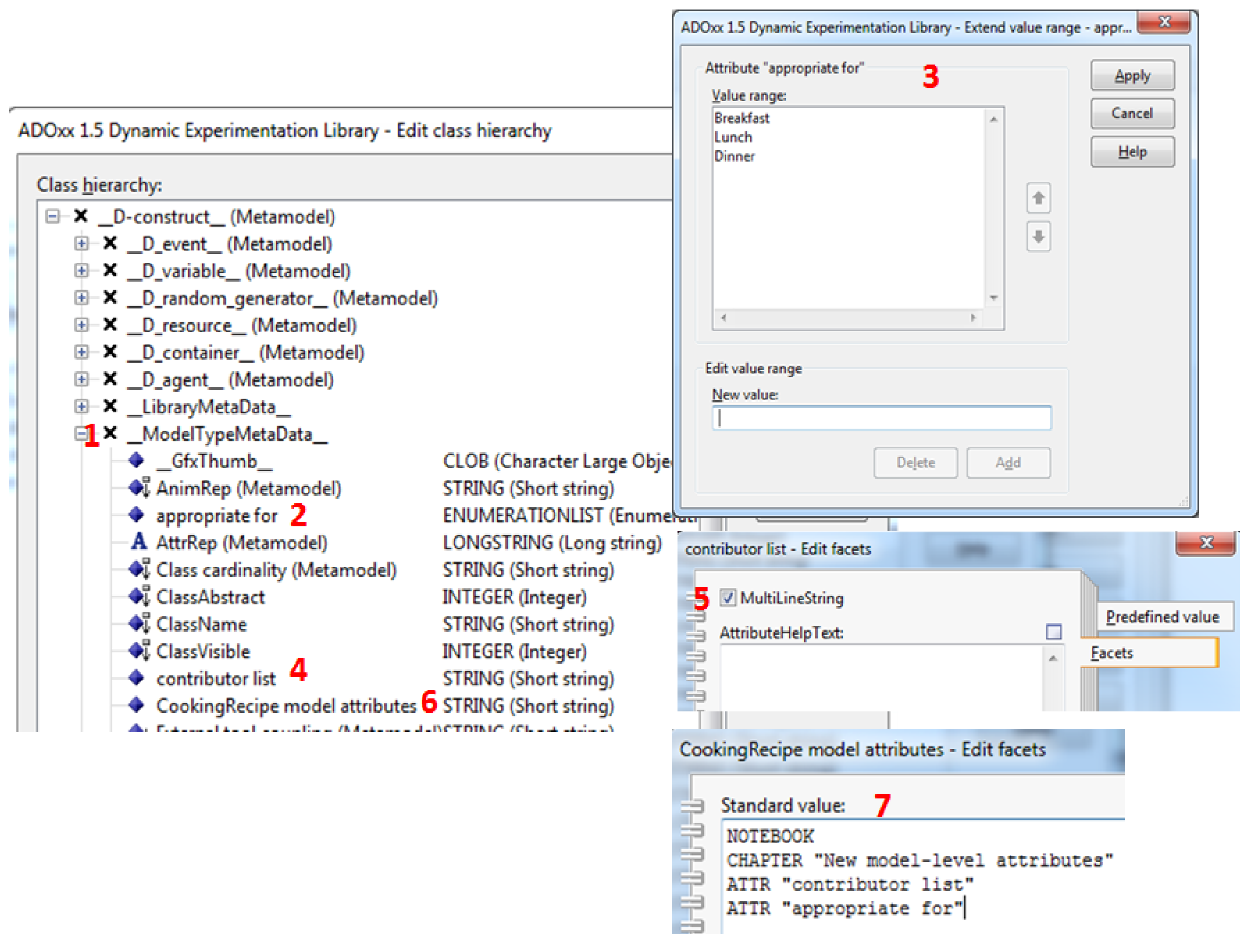


Figura 93 Creare atribute la nivel de model

Închideți ierarhia de clase și deschideți Library attributes. Adăugați la definiția tipului de model de la Cooking Recipes, parametrul attrrep care va arăta unde este păstrat codul AttrRep:

```
MODELTYPE "Cooking Recipes" attrrep:"CookingRecipe model attributes"
INCL "CookingStep"
INCL "Start"
INCL "Stop"
INCL "followed By"
```

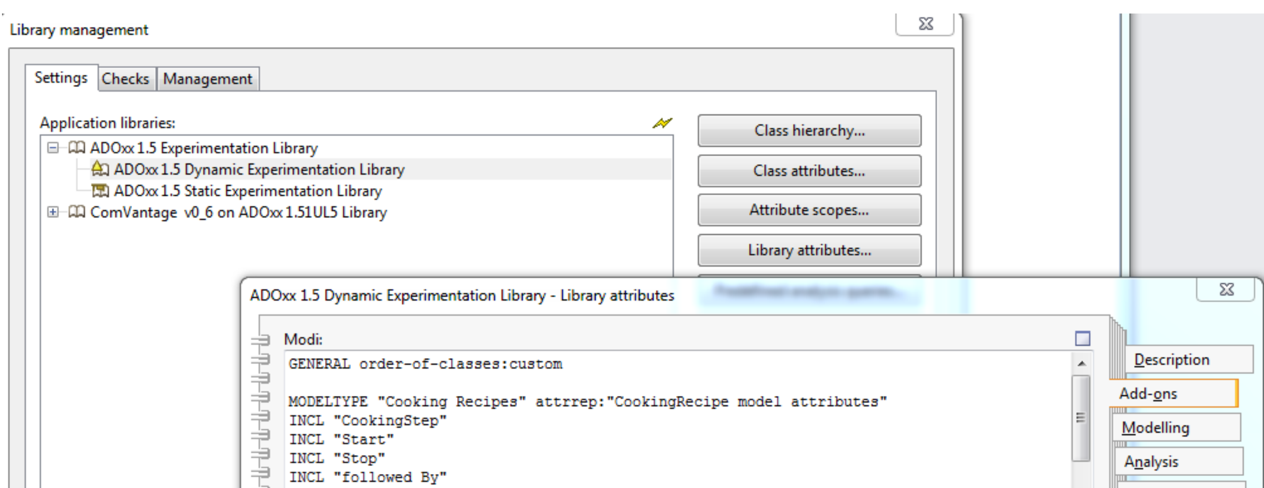


Figura 94 Specificarea la un tip de model că folosim atribute de model

Vom mai face încă o modificare la nivel de model. Este util să avem pictograme distincte pentru fiecare tip de model. Implicit toate modelele au aceeași pictogramă în fața acestora când sunt enumerate în folderele din instrumentul de modelare. Poate fi folositor să avem un indiciu vizual care modele sunt modele pentru gătit și care sunt modele pentru resurse (uneori doar numele modelului nu ne este suficient de folos).

Pentru a avea pictograme personalizate pentru fiecare tip de model, trebuie să importăm în ADOxx pictograme pe 16 pixeli. Acestea pot fi create cu orice instrument software (de ex. Paint). Sau puteți să căutați în Google ”16x16 icons” și ar trebui să găsiți colecții mari de pictograme.

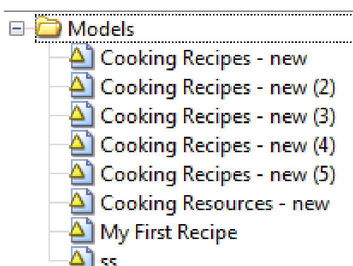


Figura 95 Lista cu modelele create în instrumentul de modelare

Vom folosi două pictograme 16x16 găsite pe Web pentru a distinge între cele două tipuri de modele după cum se vede în figura următoare. Nu vom folosi imagini mai mari deoarece nu pot fi redimensionate!



Figura 96 Particularizarea listei cu modelele create prin inlcuderea unei pictograme

Cele două pictograme trebuie să fie importate în ADOxx folosindu-vă de aceeași modalitate ca la importurile de fișiere anterioare (Extras-File Management-selecție library - Import). după import folosiți Library Attributes pentru a extinde definiția tipurilor de modele cu parametrul bitmap (fișierele au fost importate cu numele ”a.gif” și ”b.gif”:

```
MODELTYPE "Cooking Recipes" attrrep:"CookingRecipe model attributes" bitmap:"db:\\b.gif"
INCL ................................
MODELTYPE "Cooking Resources" bitmap:"db:\\a.gif"
INCL ........................
```

Porniți instrumentul de modelare și remarcați următoarele:

- noile pictograme în lista de modele
- faceți click dreapta în interiorul unui model de tip Cooking Recipes și veți observa noile atribute de la nivelul modelului: este posibil să scrieți pe mai multe linii în lista cu persoanele care au contribuit (contributor list); este posibil să alegeți mai multe opțiuni în atributul ”appropriate for”.

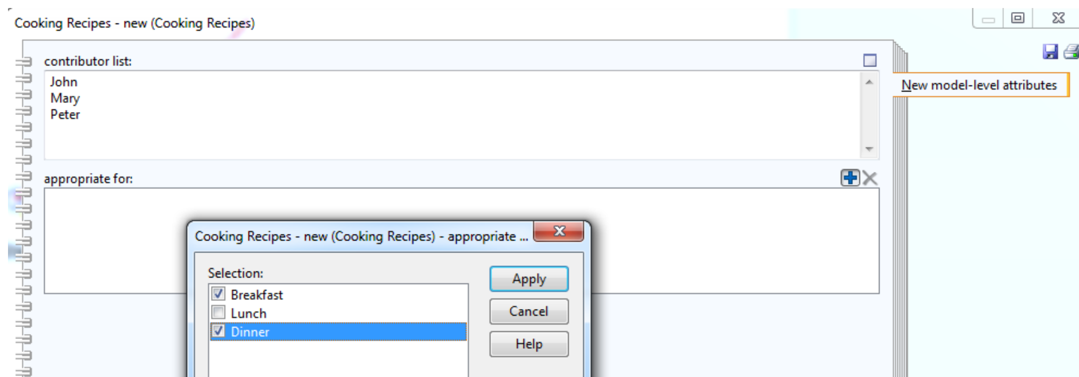


Figura 97 Atributele la nivel model disponibile la click dreapta într-un model Cooking Recipes

În continuare, vom lega cele două tipuri de modele pe care le-am creat. Un pas din procesul de gătit ar trebui să ofere hiperlegături către:

- un ingredient de orice tip (simplu sau complex), care ar fi necesar în fiecare pas de gătit
- un obiect pentru gătit necesar la fiecare pas

Limitarea la un obiect a fost aleasă pentru simplificare. Este posibil să se definească orice număr de hiperlegături dar dacă sunt mai multe, atunci nu s-ar ajunge direct la destinație ci întâi ar afișa o listă cu posibilele destinații la care s-ar putea alege. Mai târziu vom extinde exemplul pentru a oferi la unul din pași mai multe hiperlegături către mai multe ingrediente. Trebuie luat în considerare că dacă sunt necesare mai multe ingrediente la unul din pași, atunci este posibil ca modelul să nu fie suficient de detaliat! Dacă trebuie să creați mai multe hiperlegături pentru același element probabil trebuie să descompuneți pasul în alții mai mici!

Pentru a crea hiperlegături, selectați conceptul CookingStep, apoi New attribute și definiți ambele atribute (“requires ingredients”, “requires instruments”) de tip INTERREF. Figura următoare descrie cum trebuie configurate fiecare din ele:

1. selectați “requires instruments”
2. alegeți Facets
3. click pe butonul de detalii corespunzător pentru AttributeInterRefDomain
4. indicați numărul maxim de destinații pentru un link (1)
5. indicați dacă destinația linkului va fi un element al modelului (o instanță) sau un model în întregime
6. apăsați Add pentru a defini o posibilă destinație. Se va deschide o fereastră nouă
7. selectați tipul modelului de care aparține destinația (Cooking Resources)
8. selectați conceptul destinație (CookingIngredient)
9. indicați câte ingrediente pentru gătit pot fi selectate ca destinație (1)
10. apăsați Apply și observați destinațiile posibile adunate într-un tabel. Repetați de la pasul 6 pentru a adăuga ComplexIngredient ca posibilă destinație, și acesta în număr maxim de 1 (chiar dacă sunt permise concepte diferite pentru destinație doar una dintre acestea va putea fi legată într-un hiperlink, datorită pasului 4)
11. același rezultat ar putea fi obținut și prin scrierea unui fragment de cod ce poate fi observat în zona (11)

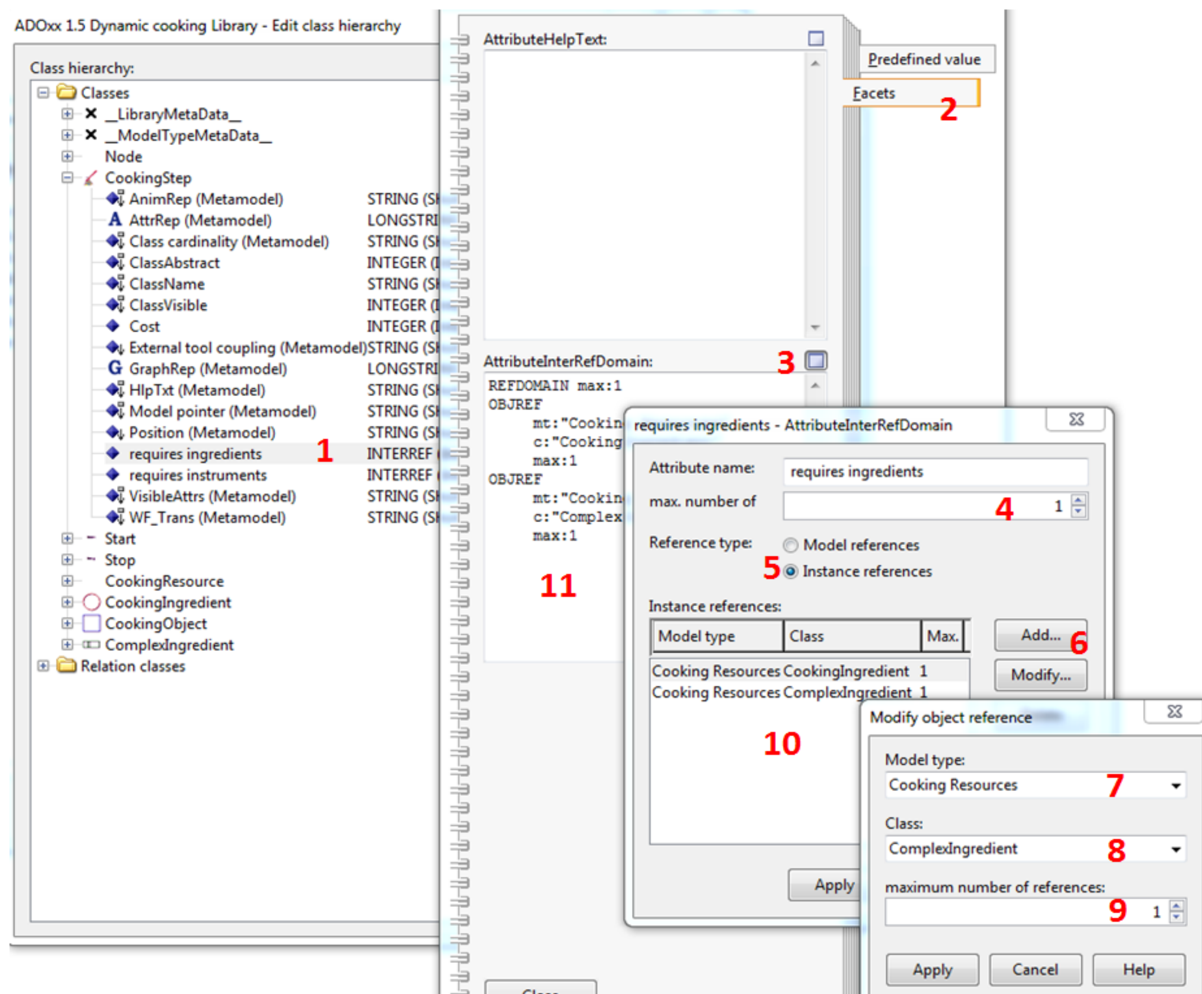


Figura 98 Creare atribute de tip hiperlegături

Faceți același lucru și pentru "requires instruments", cu deosebirea că acesta ar trebui să aibă ca destinație DOAR conceptul CookingObject (de asemenea cu un număr maxim de o destinație).

Faceți ambele hiperlegături vizibile prin includerea lor în definiția AttrRep de la CookingStep:

```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"
ATTR "requires ingredients"
ATTR "requires instruments"
```

Reveniți în instrumental de modelare și faceți dublu-click pe un pas CookingStep:

1. observați noile atribute. Fiecare dintre acestea are anumite butoane în partea din dreapta a slotului corespunzător proprietății
2. Semnul + deschide o fereastră unde puteți stabili destinația hiperlegăturii
3. selectați prima dată modelul destinație (dacă nu apare în listă, s-ar putea să trebuiască să îl deschideți prima dată; în funcție de ceea ce selectați la butonul View, este posibil să afișeze doar modelele deschise)
4. din modelul destinație, selectați obiectul destinație. Observați că pentru "requires ingredients" este posibil să se lege fie un CookingIngredient, fie un ComplexIngredient
5. după selectarea destinației, aceasta este afișată în zona (5)
6. apăsați Apply
7. dacă mai târziu doriți să eliminați hiperlegătura, folosiți pictograma (7)
8. dacă doriți să urmați hiperlegătura, folosiți pictograma (8); vă va duce către destinație.

Faceți același lucru pentru hiperlegătura “requires instruments”. Aceasta ar trebui să lege un pas de un CookingObject.

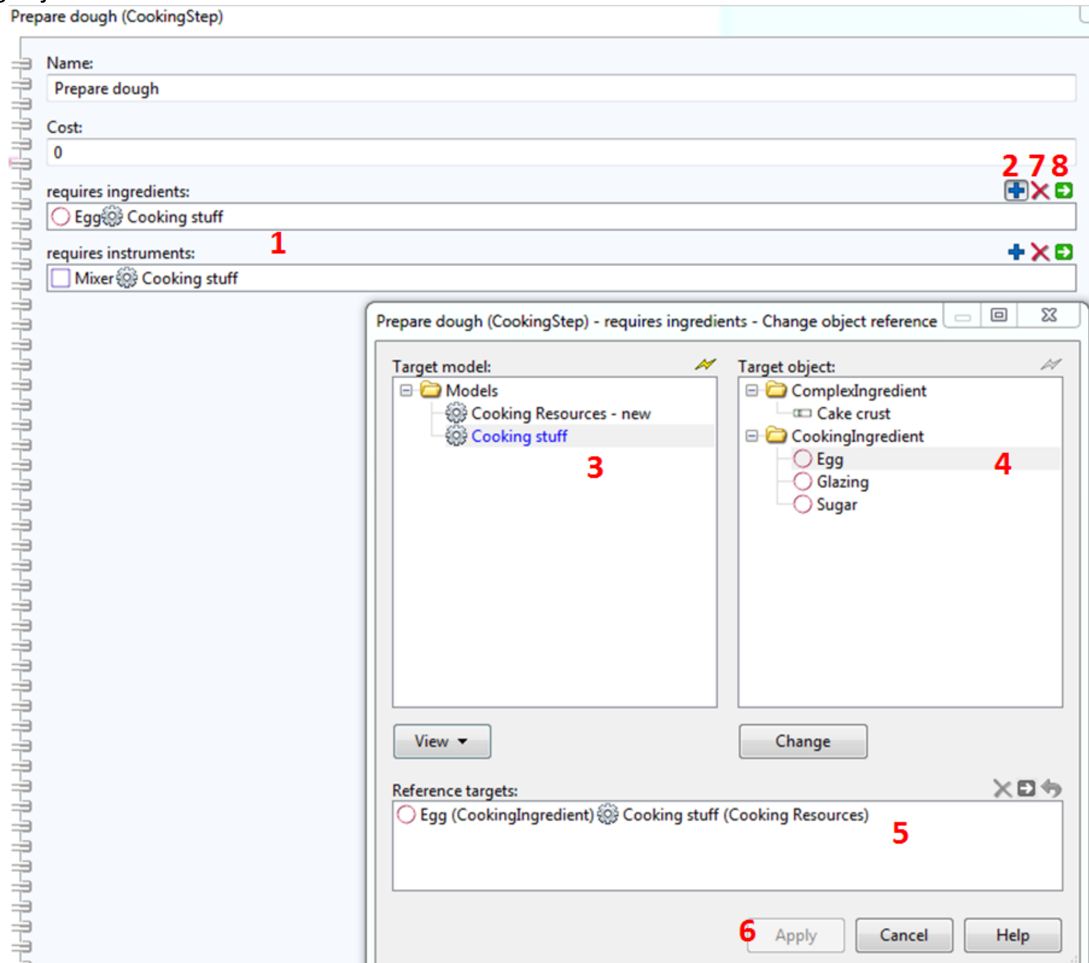


Figura 99 Creare de legături folosind atributele hyperlink create

Hiperlegăturile ar putea fi mult mai folositoare dacă utilizatorul ar putea da click direct pe simbolul grafic pentru a naviga către modelul destinație. Pentru aceasta, codul GraphRep ar trebui extins cu un element HOTSPOT. Înlocuiți GraphRep de la CookingStep cu următorul cod (sau în loc să înlocuiți vechiul cod, comentați-l):

```
GRAPHREP

FILL color:royalblue
RECTANGLE x:-1.5cm y:-1cm w:3cm h:2cm
FONT "Arial" h:14pt color:red
ATTR "Name" y:1.5cm w:c

AVAL ing:"requires ingredients"
IF (LEN ing)
  FILL color:lightblue
  ELLIPSE x:1.5cm y:-1cm rx:0.35cm ry:0.25cm
  HOTSPOT "requires ingredients" x:1.25cm y:-1.35cm w:0.7cm h:0.5cm text:"jump to ingredient"
ENDIF

AVAL tool:"requires instruments"
IF (LEN tool)
  FILL color:green
  ELLIPSE x:-1.5cm y:-1cm rx:0.35cm ry:0.25cm
  HOTSPOT "requires instruments" x:-1.75cm y:-1.35cm w:0.7cm h:0.5cm text:"jump to tool"
ENDIF
```

Remarcați că GraphRep are 3 părți:

- în primul rând simbolul de bază este definit ca fiind un dreptunghi albastru cu numele așezat sub acesta
- apoi se preia valoarea atributului ”requires ingredients”. Testăm dacă este gol (LEN este lungimea șirului, astfel testul verifică dacă lungimea este diferită de zero). Dacă nu este gol, se va desena o mică elipsă în colțul din dreapta sus al dreptunghiului și o zona HOTSPOT este definită deasupra suprafeței elipsei (zona hotspot este dreptunghiulară astfel nu este o suprapunere perfectă 100%; un text sugestiv este atașat pentru a sugera utilizatorului unde va fi dus de hotspot). Am putea extinde imaginea grafică HOTSPOT în orice modalitate permisă de sintaxa GraphRep, de exemplu pentru a afișa numele adresei linkului în cadrul zonei hotspot.
- faceți același lucru și pentru ”requires instruments” , dar definiți elipsa și zona hotspot în colțul din stânga sus.

Deschideți instrumentul de modelare. Observați în figura următoare:

1. cele două zone hotspot și textul sugestiv care apare când cursorul este deasupra zonei hotspot. Faceți click pe un hotspot pentru a trece către destinație.
2. observați butoanele înainte/înapoi care ne permit să ne deplasăm între modele

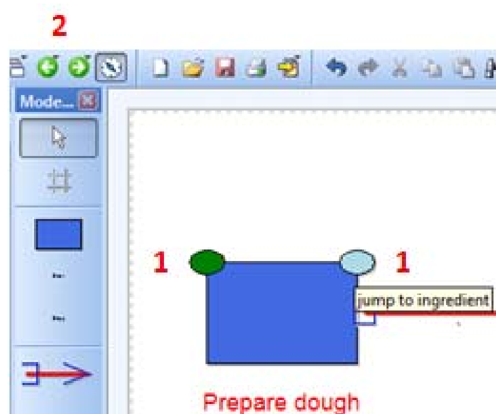


Figura 100 Modificarea reprezentării grafice ca legăturile să fie accesibile la click

În continuare vom crea un atribut complex care va arăta ca un tabel cu mai multe câmpuri. Pentru a include atribute tabulare, întâi trebuie să creăm un tip special de clasă (record class) unde vom defini structura tabelului. Acest tip de clase nu pot fi create în ierarhia conceptelor, ele trebuie create la nivel de library.

1. selectați library (dar nu partea Dynamic!)
2. deschideți ierarhia Class. Observați că la nivel library nu avem o ierarhie a conceptelor doar două tipuri de clase
3. Apăsați New și selectați Record class. Dați-i numele ”Ingredient Quantities”

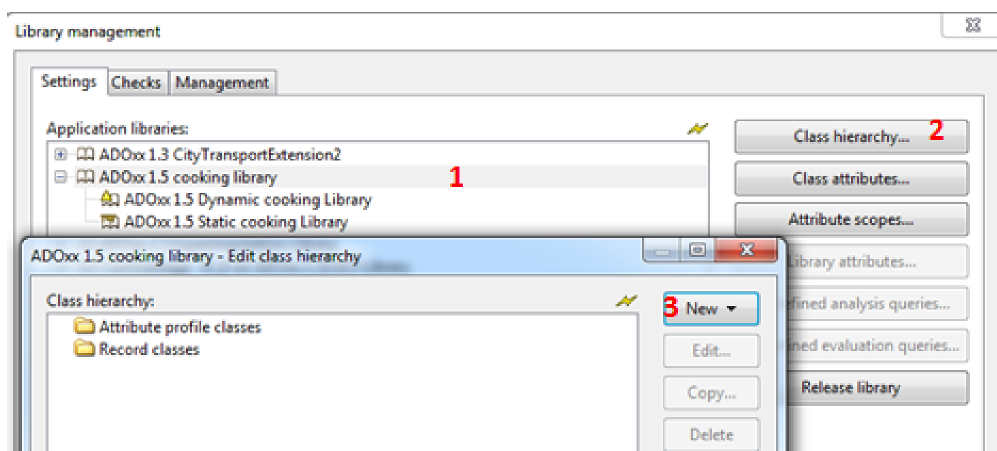


Figura 101 Creare atribut complex

O clasă record definește structura unui atribut tabel. De aceea toate câmpurile din tabel trebuie să fie definite ca atribute ale clasei record. Faceți click dreapta pe ”Ingredient Quantities” și creați două atribute:

- Ingredient, de tip INTERREF (observați că hiperlegăturile pot fi incluse în tabele!)
- Quantity de tip INTEGER

Pentru Ingredient, definiți destinația link-ului: maxim 1 destinație, referință către o instanță, tipul de model ”Cooking Resources”, clasa destinație ”CookingIngredient” (maximum 1).

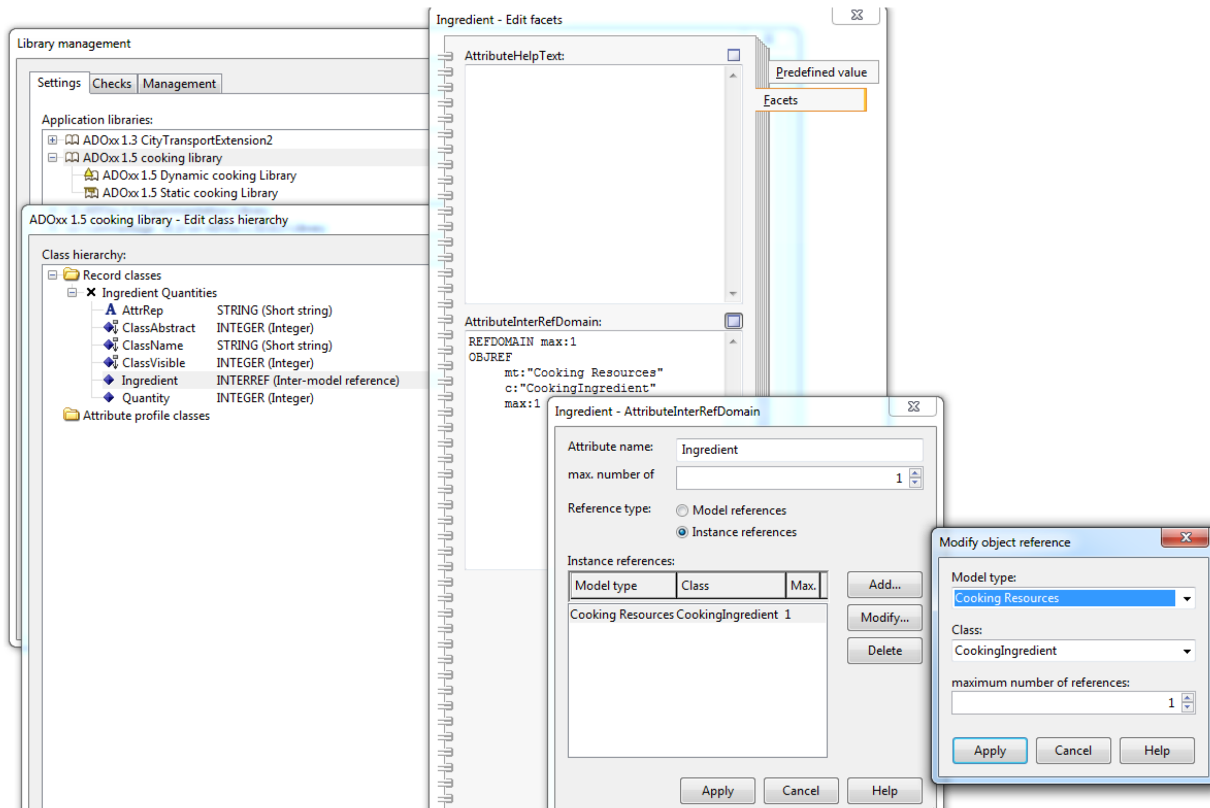


Figura 102 Stabilire structură atribut complex

La fel ca în cazul conceptelor obișnuite, este nevoie să includeți atributele în codul AttrRep al clasei record, altfel nu vor fi vizibile în tabel. Scrieți următorul fragment de cod în AttrRep de la ”Ingredient Quantities”:

```
NOTEBOOK
CHAPTER "Description"
ATTR "Ingredient"
ATTR "Quantity"
```

Apoi este nevoie să includeți acest tip de tabel în seturile de proprietăți de la CookingStep. Reveniți la ierarhia conceptelor, faceți click dreapta pe CookingStep și selectați New attribute. Numiți noul atribut ”requires ingredient quantities” și declarați-l de tip RECORD. Se va deschide o nouă fereastră în care puteți să declarați structura tabelului care va fi folosită – de ex. clasa record care tocmai ați creat-o (“Ingredient Quantities”).

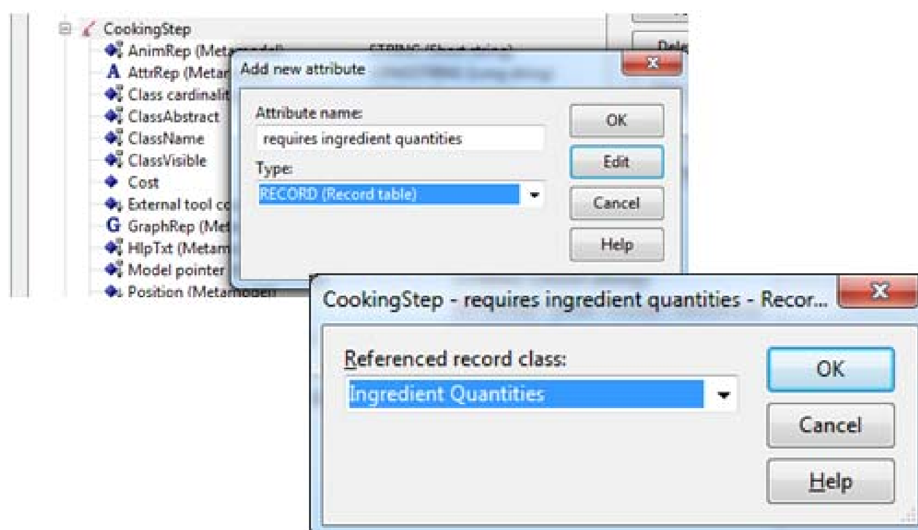


Figura 103 Includerea atributului complex într-un anumit concept

În cele din urmă trebuie să faceți tabelul vizibil în setul de proprietăți de la CookingStep, incluzându-l în AttrRep:

```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"
ATTR "requires ingredients"
ATTR "requires instruments"
ATTR "requires ingredient quantities"
```

Fiți atenți la diferența între atributul ”requires ingredient quantities” și clasa ”Ingredient Quantities”! Atributul aparține conceptului CookingStep și preia structura de tabel de la ”Ingredient Quantities”. Aceeași structură de tabel ar putea fi refolosită de alte atribute în alte concepte! ”Ingredient Quantities” nu este un atribut este doar un grup reutilizabil de atribute prezentate ca și câmpuri într-un tabel! Orice atribut de tip RECORD ar putea refolosi această structură. Cu alte cuvinte, **clasele record sunt o alternativă pentru moștenire** – de ex. acestea permit refolosirea unor proprietăți fără să le moștenească de la vreo clasă părinte din ierarhia de concepte (acesta este și motivul pentru care sunt create în afara ierarhiei de concepte!).

Deschideți instrumentul de modelare și faceți dublu-click pe un pas CookingStep. Veți observa atributul tabel și prima linie ne va permite să creăm hiperlegături către ingrediente pentru gătit disponibile în alte modele (”measurement unit” ar putea fi necesar, dar dorim să păstrăm exemplul simplu). Remarcați următoarele:

- Puteți atribui mai mult de un ingredient la un pas din procesul de gătit precum și o anumită cantitate pentru acel ingredient specific, după cum este folosit în acel pas! Aceasta înseamnă că practic stabilim o relație de la m-la-n între pașii de gătit și ingrediente iar această relație are propriul atribut (cantitate)
- Numele ingredientului are cam prea multă informație inclusă – include clasa, numele modelului destinație, tipul modelului destinație. O vom reduce doar la numele ingredientului. O altă schimbare pe care o vom face este de a afișa întreg tabelul în notația grafică pentru a face hiperlegăturile disponibile la click de mouse direct pe elementele diagramei.

Figura 104 Vizualizarea atributului complex

Avertizare: nu puteți crea hiperlegături (INTERREF) sau atribute tabel (RECORD) într-o clasă de relații (conector)

Deschideți Development Toolkit. Întâi să ne asigurăm că tabelul cu ingrediente afișează doar numele ingredientului în prima coloană. Pentru aceasta, mergeți la clasa record și modificați AttrRep de la definiția tabelului ”Ingredient Quantities”:

```
NOTEBOOK
CHAPTER "Description"
ATTR "Ingredient" format:"%o"
ATTR "Quantity"
```

Parametrul format va controla cât de mult din hiperlinkul destinație este afișat. Pot fi folosite mai multe coduri: %o este folosit pentru a specifica numele obiectului, %c este pentru numele conceptului, %m pentru numele modelului, %t pentru numele tipului de model.

Modificați în ierarhia de concepte, codul pentru GraphRep pentru CookingStep după cum urmează:

```
GRAPHREP

FILL color:royalblue
RECTANGLE x:-1.5cm y:-1cm w:3cm h:2cm
FONT "Arial" h:14pt color:red
ATTR "Name" y:1.5cm w:c

AVAL set-count-rows rowcount:"requires ingredient quantities"

IF (rowcount>0)
  FONT h:9pt
  TEXT "INGR." x:1.6cm y:-0.5cm
  TEXT "QUANT." x:3cm y:-0.5cm
  FOR i from:1 to:(rowcount)
  {
    ATTR "requires ingredient quantities" row:(i) col:"Ingredient" format:"%o" x:1.6cm y:(CM (i-1)/2)
    ATTR "requires ingredient quantities" row:(i) col:"Quantity" format:"%o" x:3cm y:(CM (i-1)/2)
    LINE x1:1.6cm y1:(CM (i-1)/2) x2:4cm y2:(CM (i-1)/2)
  }
  LINE x1:2.8cm y1:-0.5cm x2:2.8cm y2:(CM rowcount/2)
ENDIF
```

Remarcați:

- Am păstrat dreptunghiul anterior, dar am scos zonele hotspot

- AVAL acum citește numărul de rânduri din ”requires ingredient quantities” și îl va păstra în variabila internă rowcount
- Dacă rowcount este mai mare de zero, va construi un tabel cu următoarele elemente:
    - un cap de tabel, cu etichete statice (comanda TEXT) poziționate corespunzător în partea din dreapta a dreptunghiului albastru
    - o buclă *for* va genera câte un rând în tabel cu două valori colectate din coloanele Ingredient și Quantity (din row(i), rândul curent din tabel). Din nou, parametrul format va fi utilizat pentru a avea afișat doar numele linkului destinație. Poziția este aliniată cu capul de tabel, însă coordonata axei y este proporțională cu numărul de rânduri pentru a ne asigura că fiecare rând este afișat sus cel anterior. O linie simplă este trasată sub valori pentru a delimita rândurile.
    - după parcurgerea buclei FOR, o linie verticală este trasată între cele două coloane.

Reveniți în instrumentul de modelare și observați că acum tabelul este de asemenea vizibil în notație și că fiecare nume de ingredient va putea fi folosit ca o hiperlegătură nu doar în fereastra de proprietăți ci și în notație. Mai mult, remarcați că dacă structura tabelului este goală, acesta nu va fi creat deloc în notație (din acest motiv l-am inclus într-o structură IF).

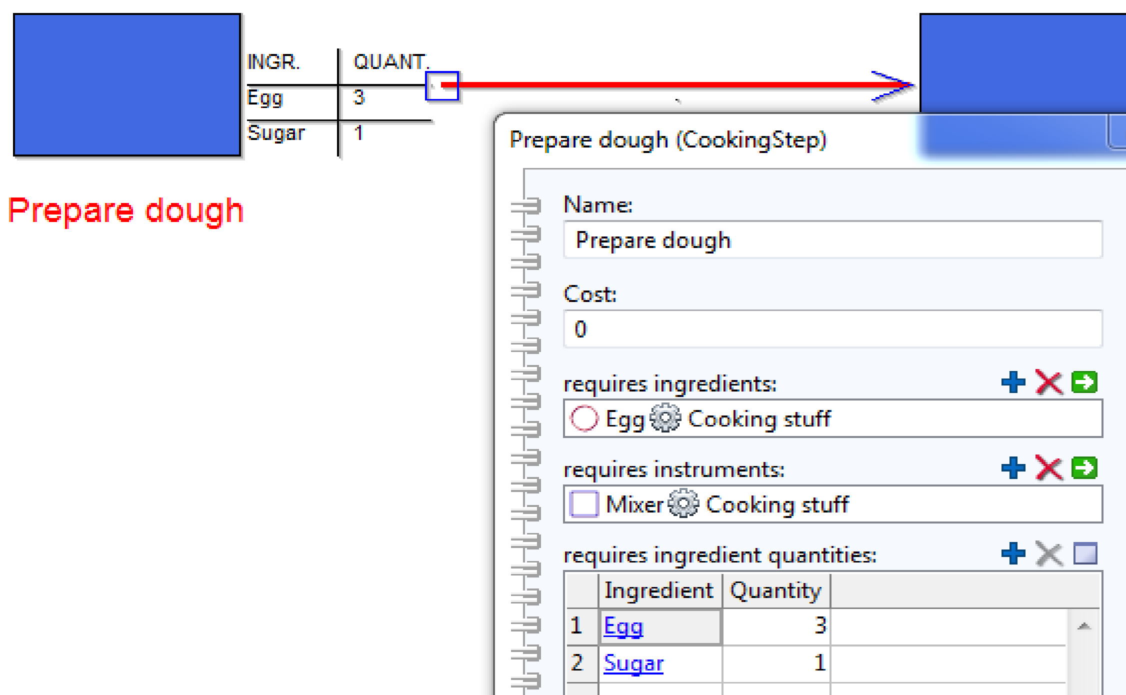


Figura 105 Includerea în reprezentarea grafică a atributului complex

În continuare vom crea un atribut calculat a cărui valori vor fi doar pentru citit deoarece acestea vor fi determinate pe baza unor calcule în funcție de alte informații păstrate în modele.

Deschideți Development Toolikit și creați pentru CookingStep un atribut nou numit ”Calculated” de tip EXPRESSION. În caseta Standard pentru acest atribut veți scrie:

```
EXPR expr:fixed:(aval("Cost")*2)
```

Cu expr:fixed, expresia va fi stabilită în Development Toolkit și valoarea acesteia se va schimba în funcție de formula de calcul. Aici formula va dubla valoarea dată de atributul Cost (accesată cu funcția aval()).

O expresie care nu este fixată va genera un atribut special care va apărea ca o expresie de construit (”expression builder”) permițând modelatorului să construiască o formulă direct în cadrul unui model, prin

folosirea unor operatori comuni și a unui set predefinit de funcții oferite de ADOxx. *Verificați Help-ul pentru Expressions pentru a obține o listă cu funcțiile disponibile.*

Creați un alt atribut în CookingStep numit ”Free formula” de asemenea de tip EXPRESSION. De această dată în caseta Standard veți scrie doar:

```
EXPR
```

În acest fel se va lăsa formula complet nerestricționată (un parametru va putea restricționa tipul valorii dat de astfel de formule). Având ambele expresii create faceți-le vizibile în AttrRep de la CookingStep:

```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"
ATTR "requires ingredients"
ATTR "requires instruments"
ATTR "requires ingredient quantities"
ATTR "Calculated"
ATTR “Free formula”
```

Reveniți în instrumentul de modelare și observați:

- slotul proprietății Calculated disponibil doar pentru citire, care afișează dublul atributului Cost
- slotul proprietății pentru formula liberă care are un buton fx la capătul din partea dreaptă. Acesta va oferi un editor pentru expresii unde se pot folosi operatori și funcții ADOxx pentru a construi calcule. Vom folosi aceeași formulă ca mai devreme însă de această dată este o formulă care a fost definită de modelator de aceea nu are o semantică bine definită (numele ”Free formula” de asemenea sugerează caracterul generic al acestei expresii). Pentru formulele libere, modelatorul ar trebui să consulte secțiunea Help pentru a se familiariza cu toți operatorii și funcțiile care pot fi folosite.

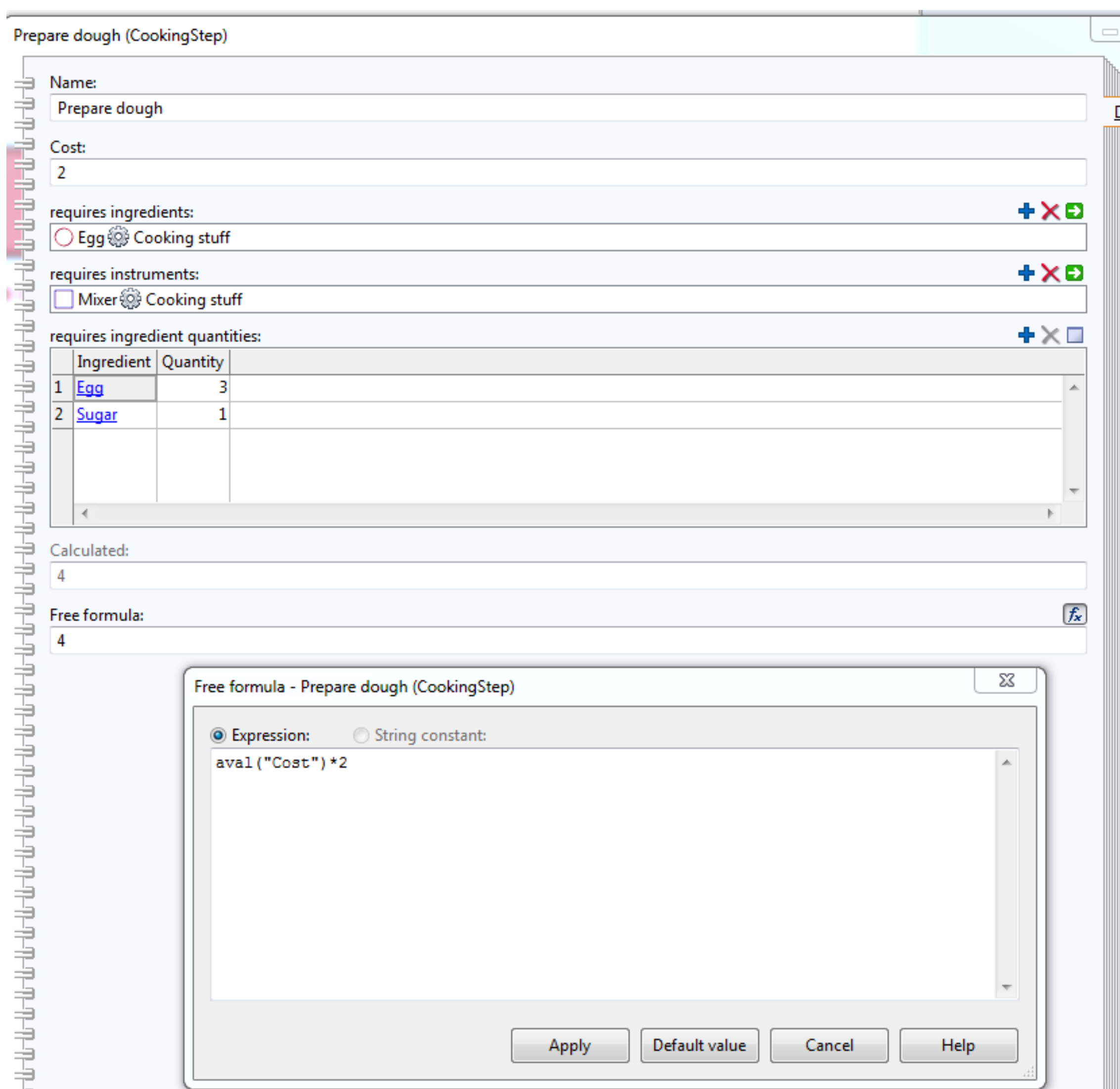


Figura 106 Vizualizarea atributului calculat și a atributului pentru definirea unei expresii de calcul

Deschideți Development Toolkit și înlocuiți valoarea EXPR de la ”Calculated” cu următoarea linie:

EXPR expr:fixed:(rasum("requires ingredient quantities","Quantity"))

Funcția rasum() calculează totalul coloanei (”Quantity”) dintr-un atribut tabel (“requires ingredient quantities”). Reveniți în instrumentul de modelare și verificați.

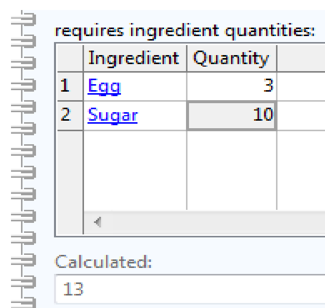


Figura 107 Schimbarea valorii atributului calculat folosind funcția de însumare pe coloană

În Development Toolkit înlocuiți valoarea de la ”Calculated” cu următoarea linie:

EXPR expr:fixed:(rcount("requires ingredient quantities"))

Funcția rcount() calculează numărul de rânduri dintr-un atribut tabel. Verificați în instrumentul de modelare.

requires ingredient quantities:

	Ingredient	Quantity
1	Egg	3
2	Sugar	10

Calculated:
2

Figura 108 Schimbarea valorii atributului calculat folosind funcția de numărare a elementelor dintr-o coloană

În Development Toolkit înlocuiți valoarea de la ”Calculated” cu următoarea linie:

```
EXPR expr:fixed:(asum(allobjs(modelid,"CookingStep"),"Cost"))
```

Funcția asum() calculează *suma valorilor pentru un atribut (”Cost”) pentru toate instanțele de la un anume concept (”CookingStep”) dintr-un model*. Observați că se bazează pe funcția allobj() care ne dă un vector cu *toate elementele de un anumit tip dintr-un anume model*. Modelul curent poate fi specificat cu constanta predefinită *modelid*. Alte constante similare sunt disponibile: objid este elementul curent din model (inclusiv conectorii); classid este clasa/conceptul curent; attrid este atributul curent; vall este valoarea curentă a expresiei. Pentru o listă completă de constante și funcții care ar putea fi folosite verificați Help-ul pentru ”Expressions”.

În instrumentul de modelare pot fi specificate diferite costuri pentru mai mulți pași din procesul de gătit, și apoi puteți observa suma calculată pentru întreg modelul. Bineînțeles o astfel de proprietate calculată nu ar avea sens să fie păstrată în fiecare element – în mod normal este păstrată în conceptul Start sau ca atribut la nivelul modelului!

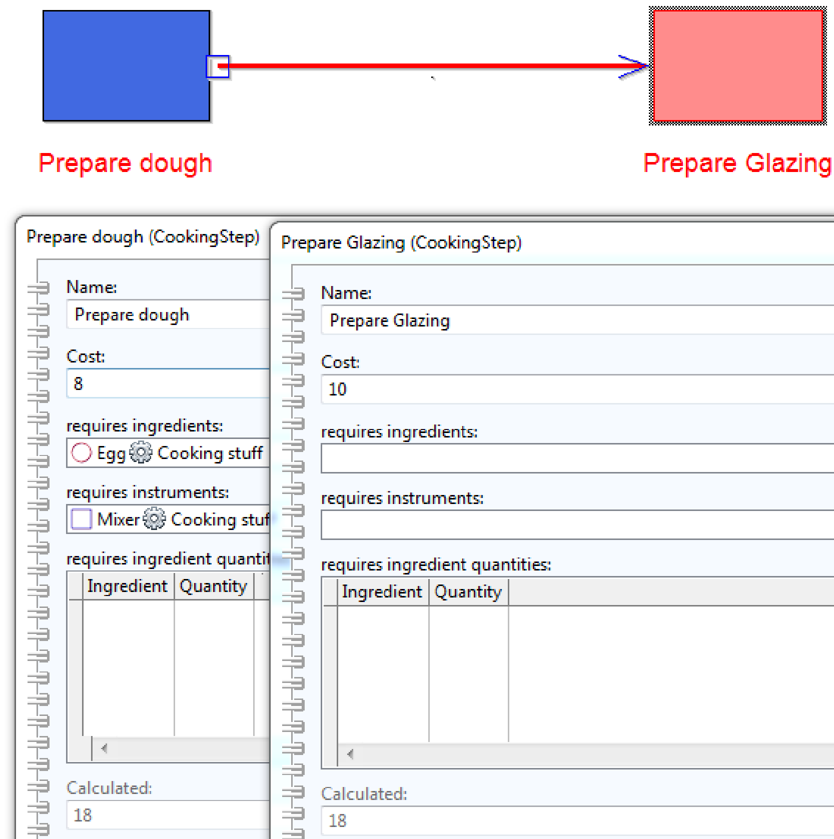


Figura 109 Schimbarea valorii atributului calculat înglobând datele de la toate obiectele din model

În continuare, vom extinde conceptul CookingStep pentru un nou tip de hiperlegătură care poate să deschidă un fișier de pe calculator sau să execute un program sau ambele (un anume fișier să fie deschis cu un anume program).

În Development Toolkit creați un nou atribut pentru CookingStep numit ”run program or file” de tip PROGRAMCALL. Acest link va fi folosit pentru a atașa fiecărui pas din procesul de gătit o anumită documentație specifică – de ex. fișiere PDF, filme care să descrie procesul de gătit sau chiar o resursă web care ar putea fi accesată printr-o adresă URL.

După ce atributul a fost creat, trebuie să îi definim EnumerationDomain:

```
ITEM "Launch Notepad" param:x
START ("notepad "+x)
ITEM "Launch Word" param:x
START ("\"C:\\Program Files\\Microsoft Office\\Office14\\Winword.exe\" "+x)
```

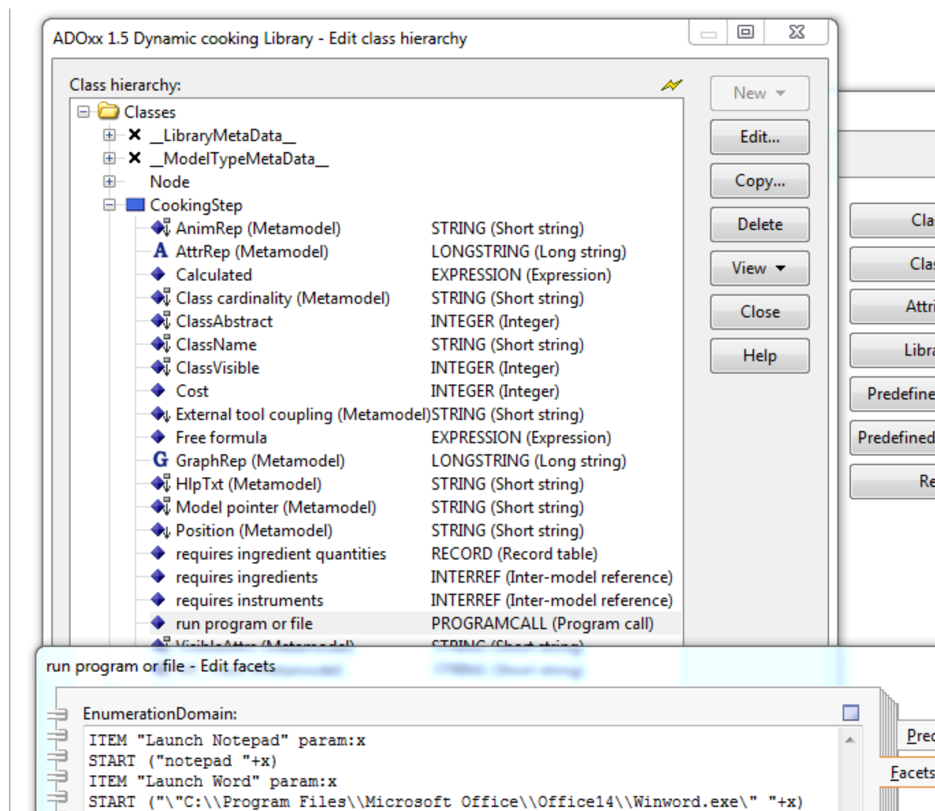


Figura 110 Crearea unui atribut de tip PROGRAMCALL

Acest fragment de cod va genera un meniu vertical cu programe care ar putea fi lansate la un pas CookingStep:

- elementul de meniu ”Launch Notepad” va executa Notepad. Va accepta un fișier ca argument (fișierul va fi selectat într-un slot dedicat pentru selecția fișierelor)
- elementul de meniu ”Launch Word” va executa Word și de asemenea acceptă un fișier ca argument
- În plus față de cele două programe, va fi inclus și un element implicit la meniu (numit ”automatically”). Acesta va permite sistemului să decidă ce program ar trebui executat în funcție de tipul fișierului selectat.

Observați că fiecare element de meniu va avea comanda START atașată de acesta care este de fapt comanda care este transmisă către linia de comandă Windows. Pentru Notepad, numele direct al acestuia ar trebui să fie suficient pentru lansare (deoarece este inclus în variabila de mediu PATH). Pentru Word însă, dacă nu dorim să o includem în PATH trebuie să îi specificăm întreaga cale cu următoarele ajustări:

caracterele backslash și ghilimelele trebuie să fie precedate de semnul backslash pentru a evita conflictul cu delimitatorii de șir ADOxx.

Apoi, vom adăuga noul atribut în AttrRep. Deoarece avem deja multe atribute vom defini un capitol nou (o pagină nouă în fereastra de proprietăți):

```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"
ATTR "requires ingredients"
ATTR "requires instruments"
ATTR "requires ingredient quantities"
ATTR "Calculated"
ATTR "Free formula"
CHAPTER "System links"
ATTR "run program or file"
```

În instrumentul de modelare găsiți noul atribut în capitolul System links de la CookingStep. Observați cele două părți de la atribut: o listă verticală (”Executable”) și un slot pentru selecția fișierelor (”Program arguments”). Butonul cu săgeată verde de la capătul din dreapta va lansa programul/fișierul selectat.

Sunt posibile mai multe combinații după cum este ilustrat în următoarele capturi de ecran:

Lăsați lista verticală cu valoarea implicită (”automtically”) și deschideți un fișier. Butonul de execuție va deschide fișierul în funcție de ce program este definit în Windows pentru tipul respectiv de fișier.



Figura 111 Selecția unui fișier text

Selectați un program din lista verticală precum și un fișier. Butonul de execuție va încerca să deschidă fișierul cu programul selectat (corespunzător cu programul definit de comanda START în Development Toolkit).

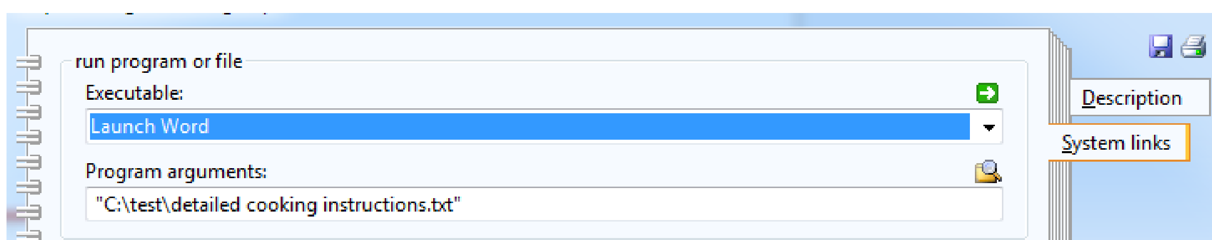


Figura 112 Selecția unui program de deschidere și a unui fișier

Selectați un program din lista verticală și lăsați slotul pentru fișier gol. Programul va fi lansat fără nici un fișier.

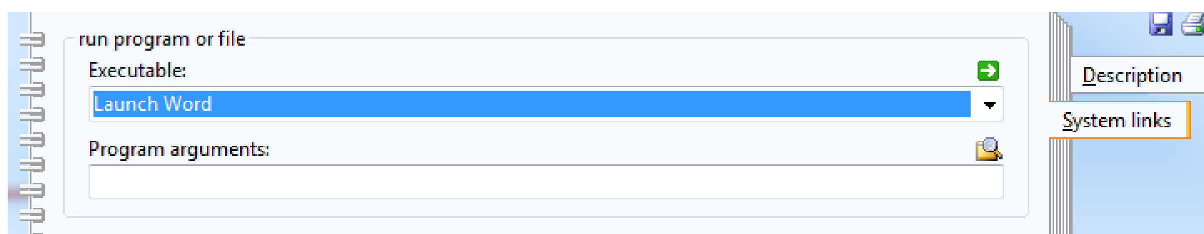


Figura 113 Selecția unui program dar nu a unui fișier

Lăsați lista verticală pe valoarea implicită și scrieți o adresă URL în slotul pentru fișiere. Browserul implicit va fi lansat și vă va duce către acel URL! Asta înseamnă că putem să legăm site-uri web relevante, videoclipuri de orice element al modelului.

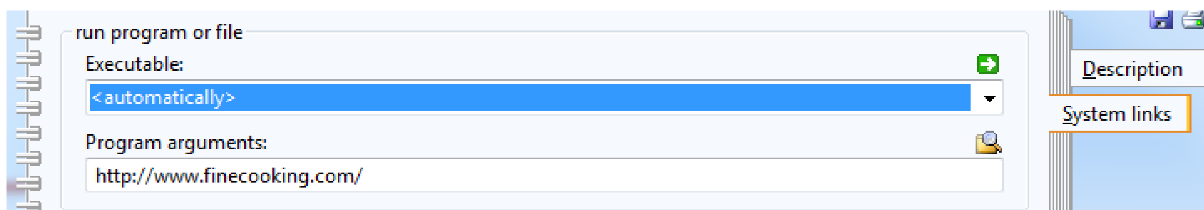


Figura 114 Scrierea unei adrese URL în loc de selecția unui fișier

Vă reamintiți că hiperlegăturile ar trebui să fie accesibile cu click direct pe simbolul grafic. De aceea vom extinde GraphRep de la CookingStep pentru a include o zonă hotspot pentru linkul către execuția programului:

```
GRAPHREP

FILL color:royalblue
RECTANGLE x:-1.5cm y:-1cm w:3cm h:2cm
FONT "Arial" h:14pt color:red
ATTR "Name" y:1.5cm w:c

AVAL pcall:"run program or file"
SET length:(LEN pcall)
SET pos:(search(pcall,"@",0))
IF ((pos) OR (length-pos-1))
  FILL color:white
  ELLIPSE x:1.5cm y:1cm rx:0.35cm ry:0.25cm
  HOTSPOT "run program or file" x:1.25cm y:0.65cm w:0.8cm h:0.4cm text:"launch application"
ENDIF
```

(puteți păstra restul codului pentru tabelul cu cantitățile ingredientelor)

Remarcați cele două variabile pos și length. Acestea nu sunt valori ale atributelor de aceea nu sunt inițializate cu AVAL ci cu SET. Pentru a înțelege raționamentul, este necesar să înțelegeți cum arată valoarea de la atributul ”run program or file” (ați putea verifica prin includerea în GraphRep a unui ATTR pentru a arăta întreaga valoare a atributului):

Valoarea atributului PROGRAMCALL are forma x@y unde x este eticheta ITEM care a fost selectată în lista verticală și y este calea către fișier ce a fost inclusă în slotul pentru selecția fișierului. Dacă rămâne selectată valoarea implicită (”automatically”) în lista verticală, x va fi un șir gol. Dacă în calea fișierului apar ghilimele (de exemplu dacă aceasta include spații) atunci și ele sunt de asemenea parte din șirul y.

De aceea, procesarea variabilelor se va face astfel:

- length va păstra numărul de caractere pentru întregul atribut
- pos va păstra poziția delimitatorului @ în cadrul valorii întregi

Zona hotspot va fi arătată doar dacă pos este diferit de zero (de ex. dacă există cel puțin un caracter suplimentar în față la @) sau dacă length-pos-1 este diferit de zero (de ex. dacă există cel puțin un caracter după@). Zona hotspot este o elipsă în colțul din dreapta jos a dreptunghiului, care va afișa un mesaj sugestiv și va permite utilizatorului să lanseze în execuție programul (sau să viziteze o adresă web) cu click direct asupra acesteia.

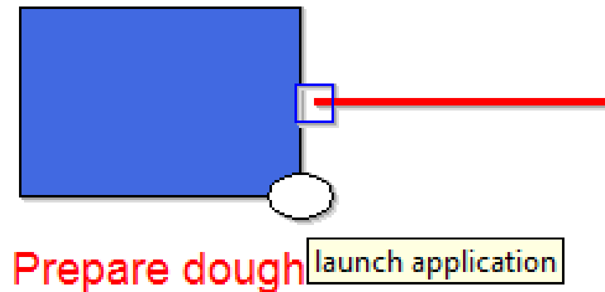


Figura 115 Includerea în reprezentarea grafică a uneui zone hotspot

În continuare, ne vom concentra asupra conceptului CookingObject din cel de-al doilea tip de diagramă. În prezent simbolul acestuia este doar un dreptunghi simplu albastru. Vom permite modelatorului să selecteze orice imagine dorește ca simbol grafic și care poate fi redimensionată în mod convenabil. Aceasta este diferit de ceea ce am făcut anterior când am impus un anumit fișier grafic pentru toate conceptele CookingStep. Aici, fiecare imagine a elementelor va putea fi aleasă de modelator.

Deschideți Development Toolkit. În primul rând creați un atribut pentru CookingObject numit “Preffered picture” de tip PROGRAMCALL. Apoi faceți-l vizibil în AttrRep:

```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Type" ctrltype:dropdown
AVAL t:"Type"
ATTR "Equipment usage" enabled:(t="Equipment")
ATTR "Preferred picture"
```

Înlocuiți codul GraphRep pentru conceptul CookingObject cu următorul:

```
GRAPHREP sizing:asymmetrical

AVAL picattr:"Preferred picture"

SET found:(search(picattr,"\"",0))
IF (found>=0)
 SET picpath:(copy(picattr,2,(LEN picattr)-3))
ELSE
 SET picpath:(copy(picattr,1,(LEN picattr)-1))
ENDIF

IF (LEN picpath)
  BITMAP (picpath) x:-1cm y:-1cm w:2cm h:2cm
ELSE
  PEN color:blue
  RECTANGLE x:-1cm y:-1cm w:2cm h:2cm
ENDIF
ATTR "Name" y:1cm w:c
```

Observați parametrul pentru redimensionare de pe prima linie. Acesta va permite modelatorului să redimensioneze simbolul în mod convenabil pentru a evita distorsionarea imaginii.

Apoi, atributul ”Preferred picture” este păstrat în picattr. Prin acest atribut vom permite modelatorului să selecteze orice fișier de pe calculator așa cum am arătat anterior. Lista verticală de selecție pentru programe va rămâne nefolosită, doar calea către fișier va conta deoarece este folosită de comanda BITMAP pentru a afișa imaginea selectată. BITMAP are nevoie doar de calea întreagă, astfel este necesar să o curățăm. Reamintiți-vă că valoarea de la atributul PROGRAMCALL va include:

- delimitatorul @ între numele de program și calea către fișier; deoarece nu vom folosi numele unui program, caracterul respectiv va fi de fapt primul din șir
- ghilimelele dacă avem spații goale conținute în calea către fișier

Prima dată, vom verifica cu search() dacă în valoarea atributului apar ghilimele (precedate de \). Funcția va returna poziția sau -1 dacă nu sunt găsite.

- dacă sunt găsite (pentru orice valoare >=0) atunci calea întreagă ar putea fi extrasă pornind de la poziția 2 (sărind peste @ și peste ghilimelele de deschidere). Numărul caracterelor extrase va fi dat de lungime minus 3 (2 pentru ghilimele și 1 pentru @)
- dacă nu sunt găsite, atunci întreaga cale începe la poziția 1 și numărul caracterelor extrase este dat de lungime minus 1

Apoi, dacă avem ceva completat la cale, comanda BITMAP este executată cu o dimensiune inițială dată. Dacă valoarea pentru path este goală, se va afișa vechiul dreptunghi.

Reveniți în instrumentul de modelare și încercați toate posibilitățile: deschideți un fișier cu o imagine a cărui cale are spații (deci cu apariția ghilimelelor), una fără spații precum și fără a completa atributul astfel încât să se afișeze doar dreptunghiul implicit.

8.8 Integrarea modelelor conceptuale cu grafuri RDF

În cele ce urmează vom unifica tehnologiile semantice discutate până aici în următoarea manieră: diagramele create cu un instrument agil de modelare vor fi exportate în format RDF pentru a fi interogate.
Presupunem următorul scenariu de business: o companie de curierat dorește să țină evidența locurilor de parcare disponibile în diverse locuri geografice unde oferă servicii și ar dori să dea angajaților săi (curierii) posibilitatea de a rezerva rapid spațiile de parcare necesare când li se atribuie o anume sarcină.
Curierii acesteia au diverse sarcini alocate ce includ activități de transport în diferite orașe.

Tehnologiile implicate sunt:

- deoarece compania de curierat are experiență în modelarea proceselor de afacere, ar prefera să folosească un instrument de modelare pentru a crea și comunica sarcinile alocate curierilor săi. ADOxx este platforma aleasă pentru instrumentul de modelare;
- datorită faptului că disponibilitatea locurilor de parcare este făcută public (pe servere RDF) de o terță companie care gestionează locurile de parcare, RDF este aleasă ca tehnologie pentru a păstra și a extrage această informație. RDF4J este folosit pentru a stoca informațiile;
- deoarece compania de curierat adoptă o politică IT de folosire a propriului dispozitiv (”bring-your-own-device”) și curierii au nevoie de mobilitate ridicată, ar prefera să aibă o aplicație pentru dispozitive mobile oferită curierilor săi.

Cerințele asumate sunt:

- un coordonator de sarcini trebuie să poată descrie prin modele a) diverse tipuri de sarcini pentru curierii săi; b) să aloce locuri de parcare în orașele în care compania oferă servicii de curierat; c) asocierea sarcinilor cu orașele unde ar trebui să fie realizate;
- un curier ar trebui să poată a) vedea lista de sarcini; b) orașele unde ar trebui să ajungă pentru realizarea sarcinilor precum și disponibilitatea locurilor de parcare din acele orașe; c) să rezerve un loc de parcare dacă este disponibil.

Figura 116 ne arată o secvență de pași ce ar trebui urmați pentru a realiza exemplul.

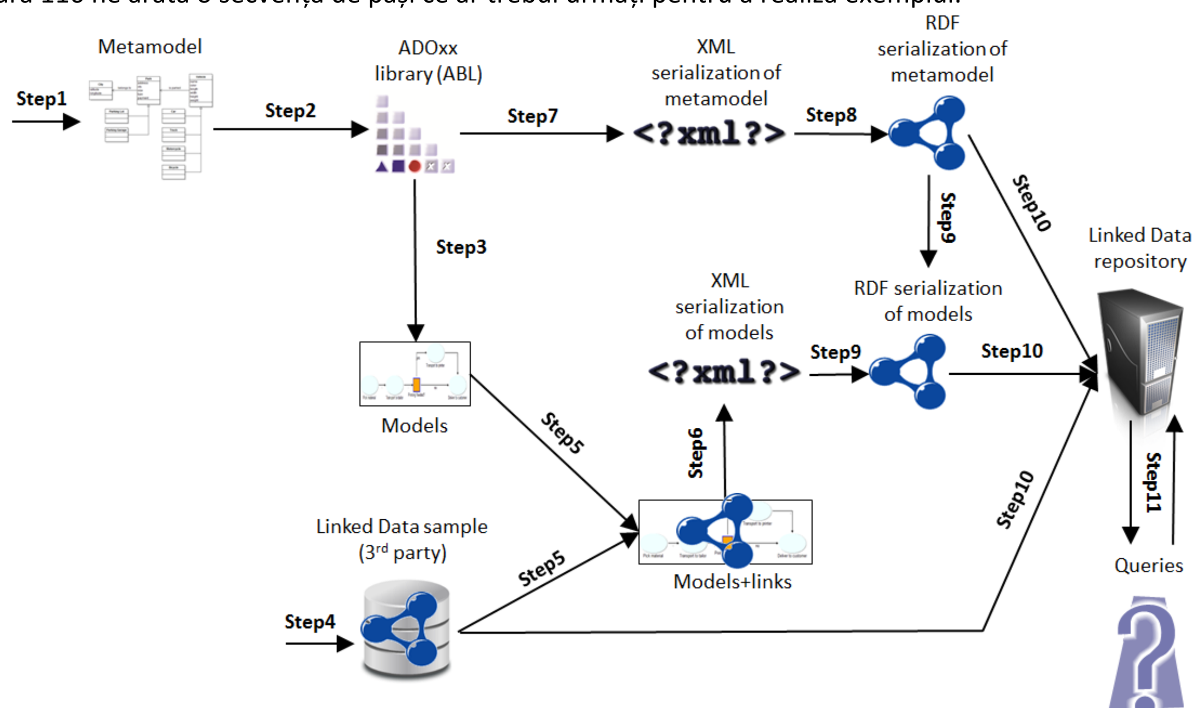


Figura 116 Secvența de acțiuni pentru realizarea exercițiului

Pasul 1. Proiectare metamodel:
- a. Intrări: scenariu/cerințe
- b. Mijloace: diagrama de clase UML

Pasul 2. Implementarea metodei de modelare:
- a. Intrări: metamodelul;
- b. Mijloace: ADOxx development toolkit;

Pasul 3. Crearea modelelor:
- a. Intrări: cunoștințe legate de scenariu;
- b. Mijloace: ADOxx modelling tool;

Pasul 4. Crearea unor date externe de test:
- a. Intrări: sisteme tradiționale (datele de test vor fi oferite în format RDF);
- b. Mijloace: orice editor text;

Pasul 5. Creare legături model-date
- a. Intrări: identificatori globali (URIs) preluați de la datelel externe;
- b. Mijloace: schimbarea atributelor URI în obiectele din modele

Pasul 6. Serializarea modelelor în XML:
- a. Intrări: modelele create în instrumentul de modelare;
- b. Mijloace: exportul XML din instrumentul de modelare;

Pasul 7. Serializarea metamodelului în XML:
- a. Intrări: ADOxx library;
- b. Mijloace: exportul XML pentru library din ADOxx development toolkit;

Pasul 8. Transformarea metamodelului în format RDF:
- a. Intrări: serializarea XML a metamodelului;
- b. Mijloace: RDFTransformer (partea de sus din interfața utilizator);

Pasul 9. Transformarea modelelor în format RDF:
- a. Intrări: serializarea XML a modelelor, reprezentarea RDF a metamodelului;
- b. Miijloace: RDFTransformer (partea de jos din interfața utilizator);

Pasul 10. Încărcare metamodel, modele și date într-un depozit RDF:
- a. Intrări: reprezentarea RDF a modelelor, metamodelului și a datelor externe,

b. Mijloace: interfața utilizator RDF4J;

Pasul 11. Exemple interogări:

a. Mijloace: interfața utilizator RDF4J;

Pasul 1. Proiectarea metamodelului

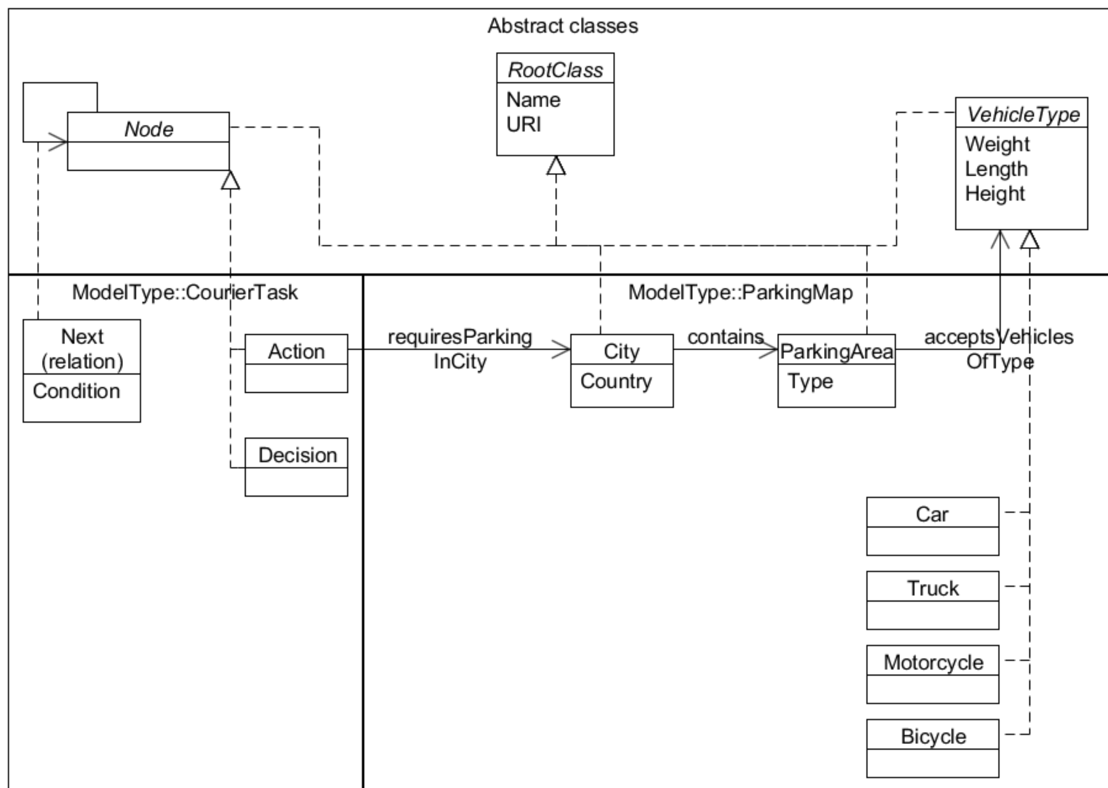


Figura 117 Metamodelul

Metamodelul (Figura 117) **arată două tipuri de modele:**

- Modelul locațiilor de parcare - *ParkingMap* ne permite să modelăm orașe, locuri de parcare, vehicule - *Cities*, *ParkingAreas* și patru tipuri de *Vehicles*
  - Orașele *(Cities)* pot conține mai multe zone de parcare (ParkingAreas); ParkingAreas pot accepta diverse tipuri de vehicule (Vehicle types) (pentru scopul acestui exercițiu cardinalitatea relațiilor este irelevantă și nu va fi considerată);
- Modelul sarcinilor de curierat - *CourierTask* ne permite să descriem o sarcină ca o secvență de acțiuni de transport necesare (*Actions)*; pe lângă acțiuni, deciziile (*Decisions)* pot fi folosite pentru a împărți secvența în mai multe căi.
  - *Next* este numele relației care leagă nodurile următoare într-o sarcină (are un atribut *Condition* pentru a descrie, după o decizie, ce anume determină alegerea unei anume căi;
  - *Actions* pot fi asociate cu *Cities* în care ar trebui să se realizeze (însemnând faptul că un curier va trebui să găsească un loc de parcare în acele orașe unde trebuie să-și realizeze sarcinile); pentru aceasta se va folosi legătura ”interref” *requiresParkinginCity* (de la *Action* din *CourierTask* către *City* în *ParkingMap*).

Observați atributul URI moștenit de la Root, ceea ce înseamnă că va fi disponibil în toate obiectele. Acesta va fi folosit pentru a lega informațiile din modele cu date externe.

Pasul 2. Implementarea metamodelului în ADOxx

Conform instrucțiunilor din capitolul precedent, transpuneți metamodelul în conceptele vizibile în Figura 118. Nu e obligatoriu să respectați simbolurile grafice din imagine, acestea nu vor influența exportul în format RDF, deoarece grafurile RDF vor stoca doar semantica diagramelor (pentru interogări), nu și forma lor grafică.

Indicăm în continuare în linii mari pașii de urmat pentru implementarea metamodelului, ca recapitulare pentru ce s-a discutat deja în capitolul precedent.

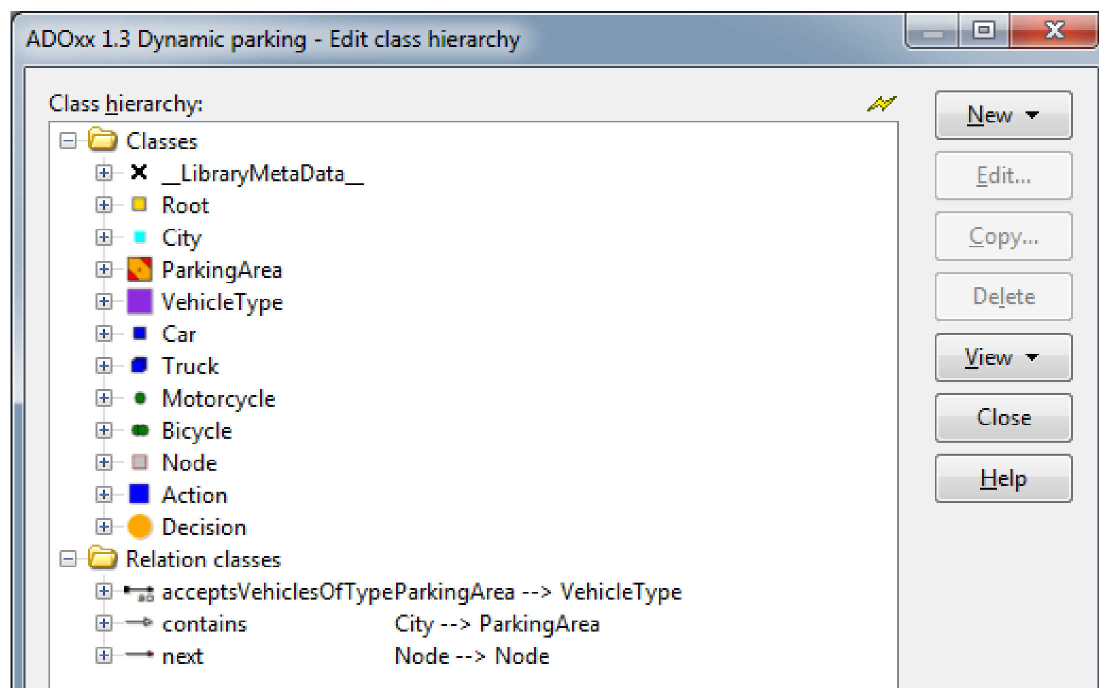


Figura 118 Implementarea în ADOxx a metamodelului

După ce ați definit conceptele împreună cu simbolurile și atributele lor, grupați-le în cele două tipuri de diagrame:

1. Deschideți componenta Library Management
2. Deschideți Settings
3. Selectați componenta Dynamic
4. Deschideți Library attributes
5. Deschideți atributul de grup Add-ons și apoi atributul Modi. Acesta conține definiția generală pentru tipurile de modele (ce concepte și ce relații sunt disponibile în fiecare tip de model):

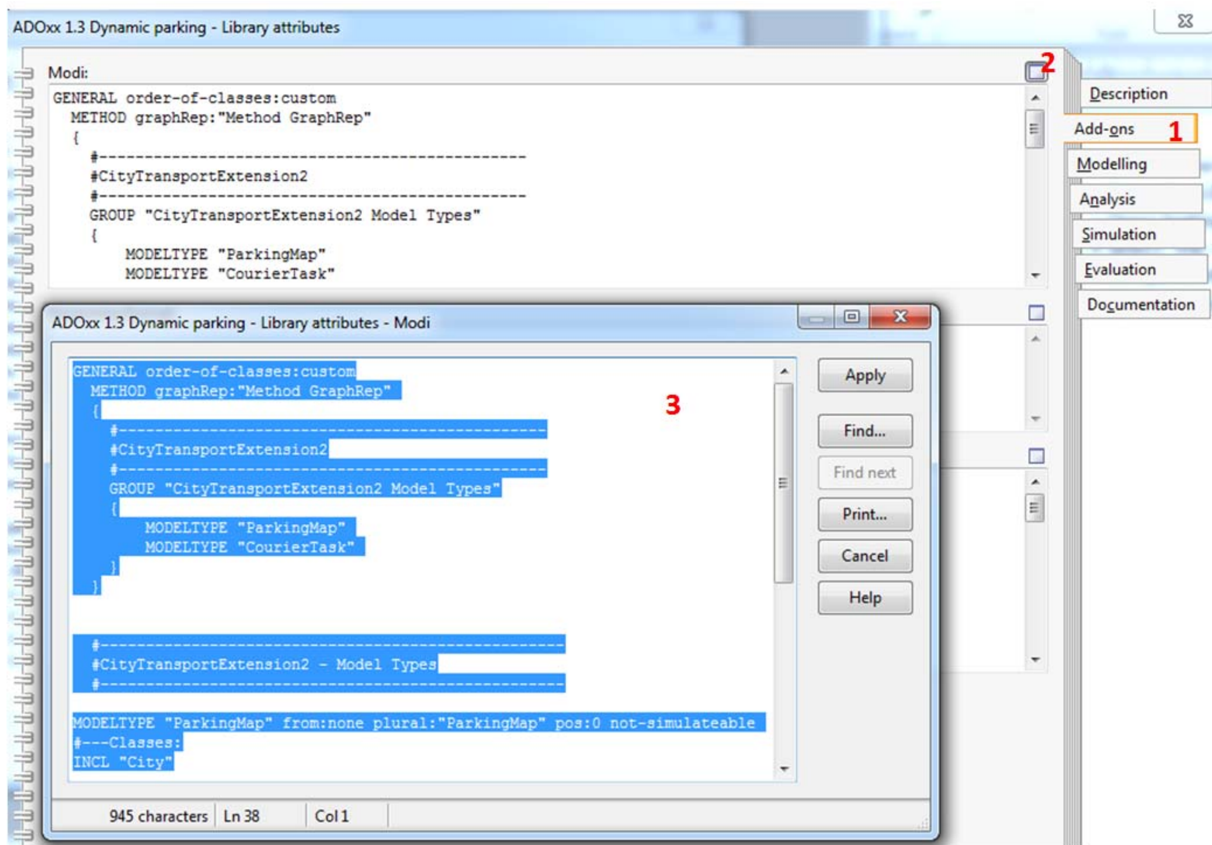


Figura 119 Definiția tipurilor de modele în atributul Modi

Codul corespunzător ar fi următorul:

```
GENERAL order-of-classes:custom
 METHOD graphRep:"Method GraphRep"
 {
  #-----------------------------------------------
  #CityTransportExtension2
  #-----------------------------------------------
  GROUP "CityTransportExtension2 Model Types"
  {
        MODELTYPE "ParkingMap"
        MODELTYPE "CourierTask"
  }
 }

 #--------------------------------------------------
 #CityTransportExtension2 - Model Types
 #--------------------------------------------------

MODELTYPE "ParkingMap" from:none plural:"ParkingMap" pos:0 not-simulateable
#---Classes:
INCL "City"
INCL "ParkingArea"
INCL "Car"
INCL "Truck"
INCL "Motorcycle"
INCL "Bicycle"
#---Relation Classes:
INCL "acceptsVehiclesOfType"
INCL "contains"
#---Modes:
MODELTYPE "CourierTask" from:none plural:"CourierTask" pos:0 not-simulateable
#---Classes:
INCL "Decision"
INCL "Action"
#---Relation Classes:
INCL "next"
#---Modes:
```

Avertisment: acest cod poate fi creat doar după ce toate clasele și toate relațiile au fost create! De aceea, conceptele și relațiile trebuie să existe în ADOxx înainte de a le organiza pe tipuri de modele.

Toate atributele trebuie să fie definite, iar apoi organizate pentru a fi afișate. Cele mai multe atribute sunt moștenite de la ADOxx (de ex. Position) dar vom crea și unele proprii (de ex. Country, URI). Atributul URI este necesar pentru a indica ce identificator dorim pentru fiecare obiect (altfel, identificatorii sunt generați în funcție de numele obiectelor și unele valori aleatoare).

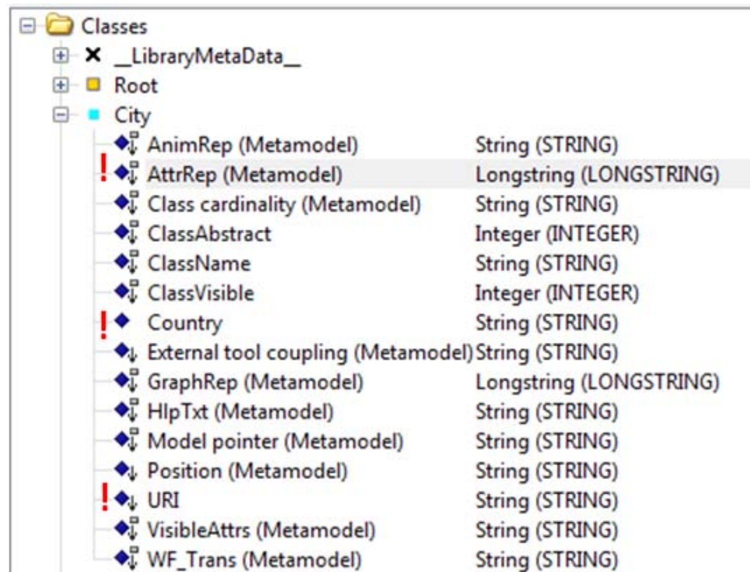


Figura 120 Atributele solicitate pentru conceptul City

După ce atributele (și tipurile acestora) au fost definite, acestea pot fi organizate pentru a fi afișate folosind atributul predefinit AttrRep:

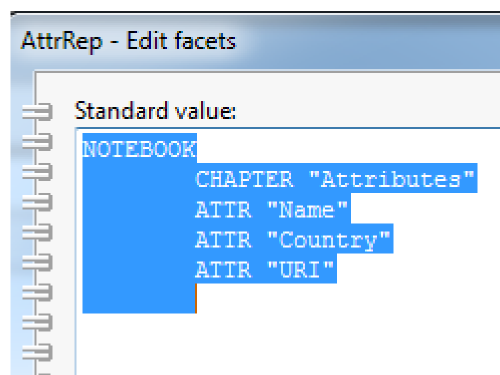


Figura 121 Stabilirea atributelor vizibile pentru conceptul City

Un tip particular de atribut este hiperlegătura spre alte modele (declarat ca INTERREF). În acest exemplu avem doar o singură hiperlegătură, cea care leagă Actions de Cities din Parking Map.

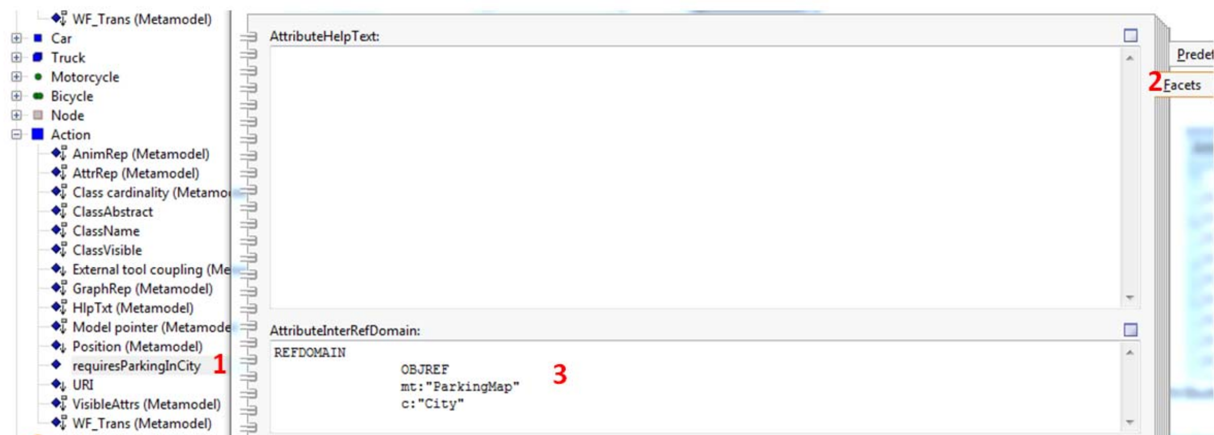


Figura 122 Crearea atributului "requiresParkingInCity" de tip INTERREF pentru conceptul Action

Pentru relații este important să definim domeniul (from) și codomeniul (to). Acestea pot să aibă și atribute, precum Condition pentru a specifica modul de realizare a sarcinilor, sau AttrRep pentru organizarea atributelor ce ar trebui afișate.

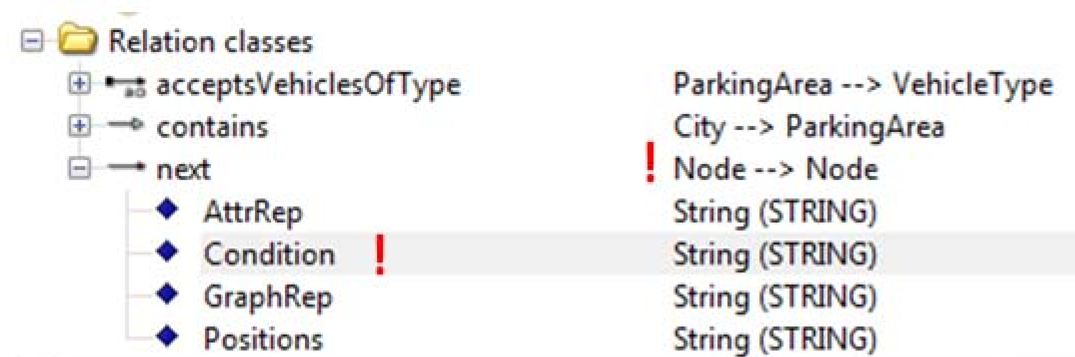


Figura 123 Crearea unui atribut pentru conectori

Un utilizator trebuie definit împreună cu datele folosite la autentificarea în instrumentul de modelare. Acești pași sunt prezentați în Figura 124:

1. Deschideți componenta *User management*;
2. Deschideți *User list*;
3. Adăugați un utilizator nou;
4. Definiți un nume de utilizator;
5. Definiți o parolă pentru utilizator;
6. Confirmați parola;
7. Selectați library pentru metoda importată;
8. Deschideți *user group list*;
9. Atribuiți utilizatorul unui grup de utilizatori (*ADOxx*);
10. Apăsați *OK*;
11. Apăsați *Add*;
12. Închideți componenta *User management*

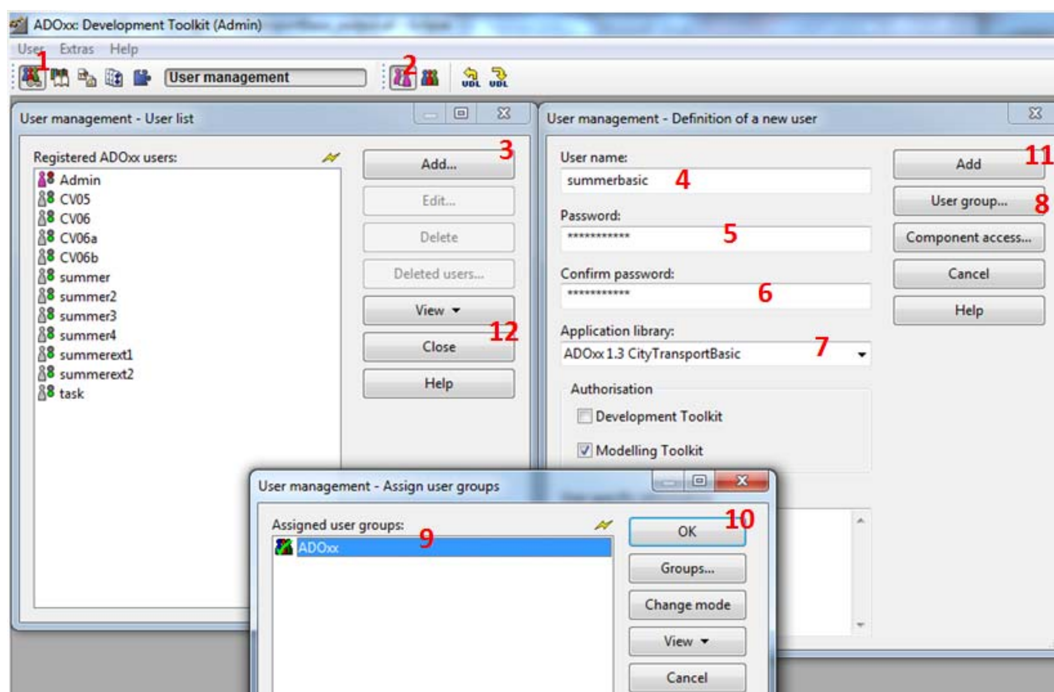


Figura 124 Pașii urmați pentru atribuirea unui utilizator existent la metoda importată

Pasul 3 Crearea modelelor

Deschideți instrumentul de modelare, folosindu-vă de datele definite în pasul anterior și creați modele precum cele din Figura 125:

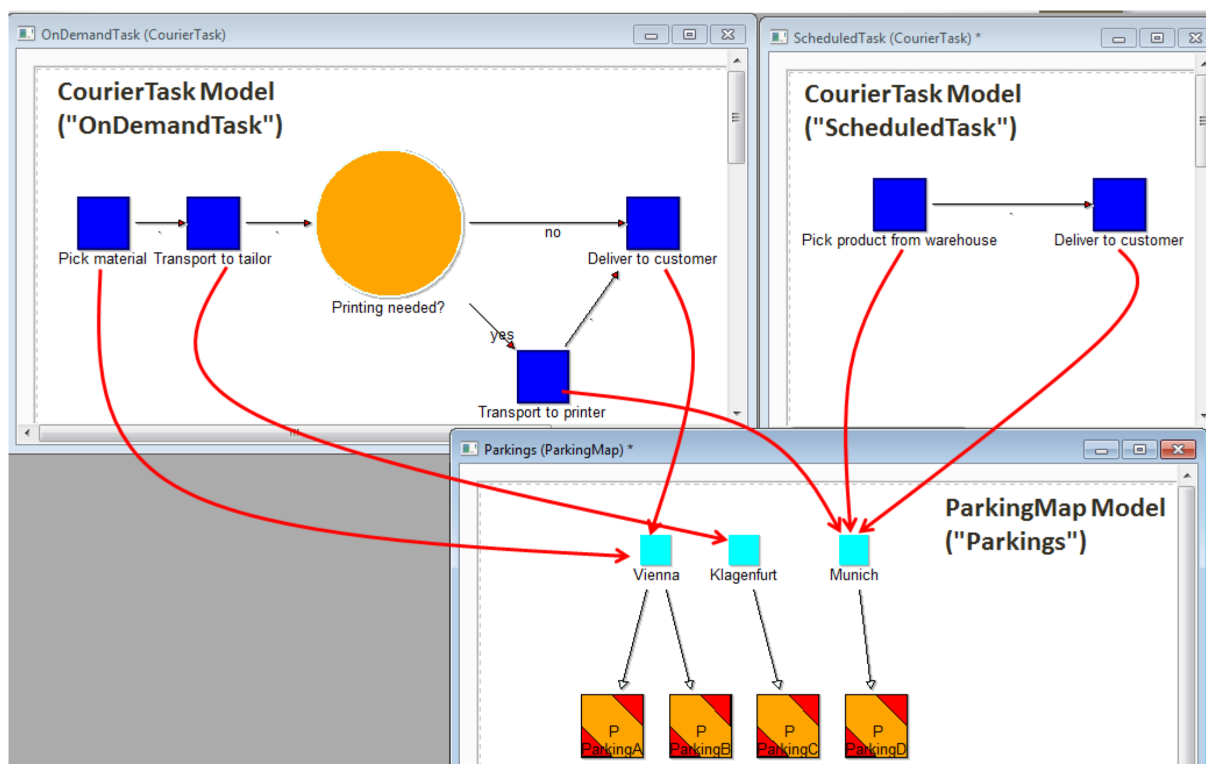


Figura 125 Modele create în instrumentul de modelare

Modelele arată două sarcini de curierat asociate (prin intermediul legăturii *requiredParkingInCity)* cu orașele unde ar solicita locuri de parcare (dintr-un model Parking map).

Pasul 4. Crearea unor date de test externe

Se presupune că datele oferite de terțe părți legate de disponibilitatea în timp real a locurilor de parcare sunt gestionate de o companie responsabilă de parcări. Datele sunt disponibile ca grafuri pentru a fi folosite în mod public și gratuit. Astfel de date sunt cele afișate în tabelul 2 așa cum ar fi în forma originală preluate dintr-o baza de date relațională:

Parking Areas	
Parking ID	**Availability**
ParkingA	2
ParkingB	0
ParkingC	4
ParkingD	0
ParkingE	3

Tabelul 1 Date în format tradițional de la terțe părți

Tabelul conține identificatorul fiecărei zone de parcare și numărul locurilor disponibile. Instrumente pentru realizarea conversiei din tabele în RDF sunt D2RQ[73], XLWrap[74], OntoRefine[75]. Rezultatele generate de o astfel de conversie sunt prezentate mai jos (se presupune că datele sunt grupate în cadrul unui identificator de graf numit http://example.org#LegacyData):

```
@prefix : <http://example.org#>.
:LegacyData
{
:ParkingA :availability 2 ; a :ParkingArea.
:ParkingB :availability 0 ; a :ParkingArea.
:ParkingC :availability 4 ; a :ParkingArea.
:ParkingD :availability 0 ; a :ParkingArea.
:ParkingE :availability 3 ; a :ParkingArea.
}
```

Pentru a evita folosirea unui adaptor, tastați direct datele în serverul RDF4J.

Pasul 5. Creare legături între modele și date

Acest pas este esențial pentru a realiza legătura dintre informația din modele și datele externe preluate din sisteme tradiționale. În RDF, conectarea datelor este realizată prin refolosirea URI-urilor ca identificatori globali, în acest fel indicând că datele se referă la ”același lucru”. Cu alte cuvinte, lucrurile care au aceiași identificatori URI sunt considerate echivalente.

Acest pas presupune că identificatorii folosiți de compania de parking care deține datele au fost făcuți publici sau puși la dispoziție către compania de curierat care deține modelele. În acest caz, legătura dintre modele și date este realizată dacă declarăm că zonele de parcare din modele (*Parkings)* sunt aceleași cu cele din baza de date relațională. Calea cea mai directă de a realiza acest lucru este de a refolosi în modele URI-urile de la zonele de parcare.

Acestea sunt cele folosite în datele de test de la pasul 4 (prefixul concatenat cu ID-ul):

[73] http://d2rq.org/
[74] http://xlwrap.sourceforge.net/
[75] http://graphdb.ontotext.com/free/loading-data-using-ontorefine.html

http://example.org#ParkingA
http://example.org#ParkingB
http://example.org#ParkingC
http://example.org#ParkingD

Identificatorii de parcare trebuie să fie scriși în atributul URI după cum se vede în Figura 126.
Notă: De fiecare dată când atributul URI este lăsat necompletat, un identificator aleator va fi generat (care nu va oferi nici o conectare). Scrierea greșită a identificatorilor va afecta legăturile între modele și date!

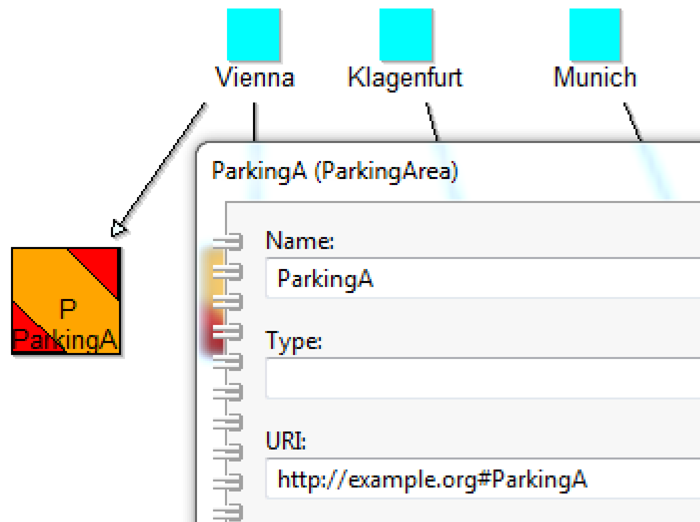


Figura 126 Crearea legăturilor dintre modele și date

Pasul 6. Serializarea modelelor în XML

Pentru instrumentul de modelare, Figura 127 indică opțiunile care ar fi folosite pentru a realiza serializarea modelelor în format XML:

1. Activați componenta *Import/Export;*
2. Selectați opțiunea *XML Export;*
3. Selectați cele trei modele (acestea vor fi păstrate ca un singur fișier XML);
4. Apăsați butonul *Export*;

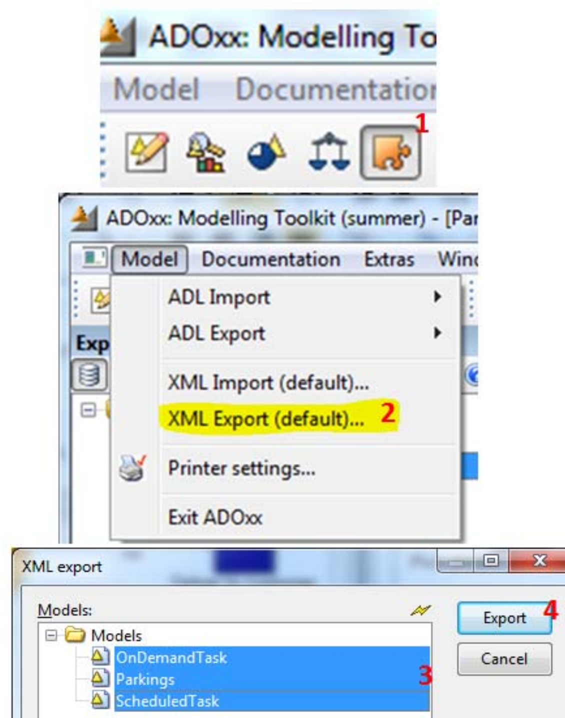


Figura 127 Serializarea XML a modelelor

Pasul 7. Serializarea metamodelului în XML

Pentru ADOxx Development Toolkit, Figura 128 indică pașii ce trebuie urmați pentru a realiza serializarea XML a metamodelului:

1. Deschideți componenta *Library Management*;
2. Deschideți tabul *Management*;
3. Selectați library pentru metoda curentă;
4. Selectați opțiunea *XML Export*;
5. Selectați folderul destinație;
6. Apăsați butonul *Export*;

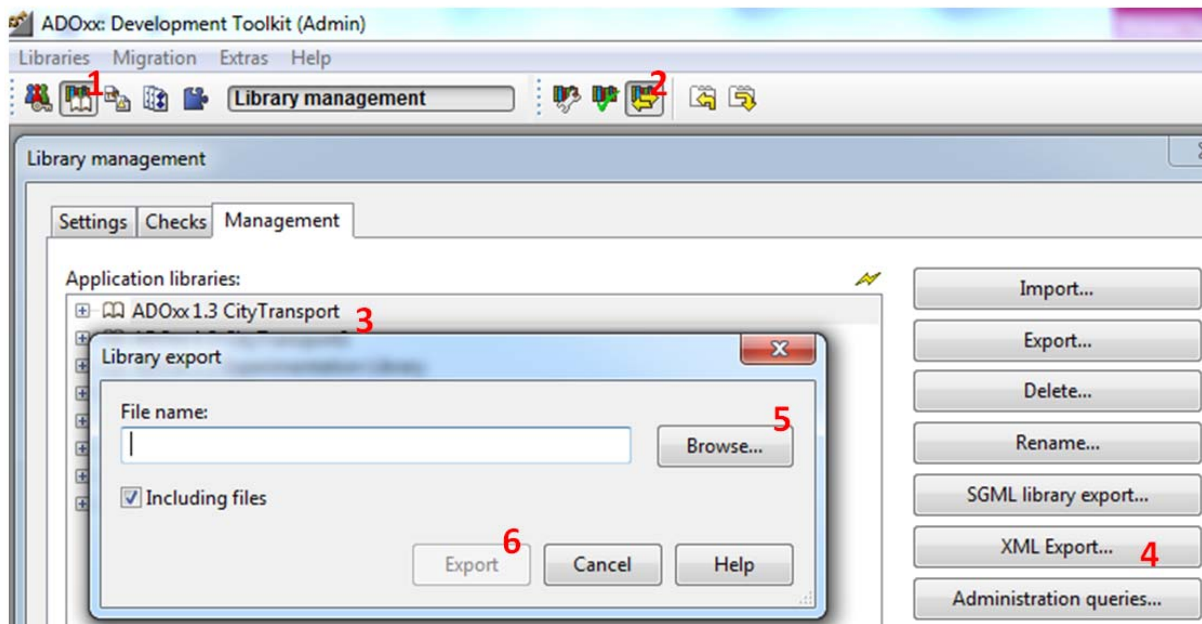


Figura 128 Serializarea XML pentru metamodel

Pasul 8. Transformarea metamodelului în format RDF

Instrumentul RDFTransformer.jar[76] trebuie să fie executat pentru a transforma exportul XML în RDF. Acesta solicită o instalare Java și trebuie să fie în același folder cu fișierul adoxml31.dtd (schema pentru serializarea XML)

La execuție, se va afișa interfața utilizator vizibilă în Figura 129. Partea de sus a ferestrei va fi folosită pentru transformarea metamodelului, în timp ce partea de jos se va folosi pentru transformarea modelelor. Pentru a încărca versiunea XML a metamodelului apăsați butonul Load din partea superioară (marcat cu 1). Va solicita specificarea unui URI corespunzător pentru partea de metamodel din exportul RDF. Acest exercițiu va presupune că acel URI este *http://example.org#metamodelgraph* (marcat cu 2).

Notă: nu scrieți greșit identificatorii! Exemplele de interogări se vor baza pe aceștia.

După ce metamodelul este încărcat, lista cu atributele găsite în clase este afișată în partea din stânga (Figura 130). Folosind butoanele +/- de mai jos, utilizatorul are posibilitatea de a filtra metamodelul astfel încât doar atributele relevante vor fi exportate ca Linked Data. Pentru scopul acestui exercițiu, vom exporta întregul conținut al modelelor, astfel nu se va aplica nici o filtrare.

[76] http://austria.omilab.org/psm/content/enterknow/info?view=home

Înainte de a transforma metamodelul, utilizatorul trebuie să indice care dintre atribute a fost folosit ca identificator global (pentru conectarea model-date). După cum s-a discutat mai devreme, atributul *URI* a fost inclus în metamodel tocmai pentru acest scop și va fi selectat aici (marcat cu 1 în Figura 130), urmat de apăsarea butonului *URI switch* (marcat cu 2) care ar trebui să schimbe atributul URI în albastru. Apoi, metamodelul trebuie să fie salvat (marcajul 3). Formatul TriG ar trebui să fie selectat în fereastra *Save*.

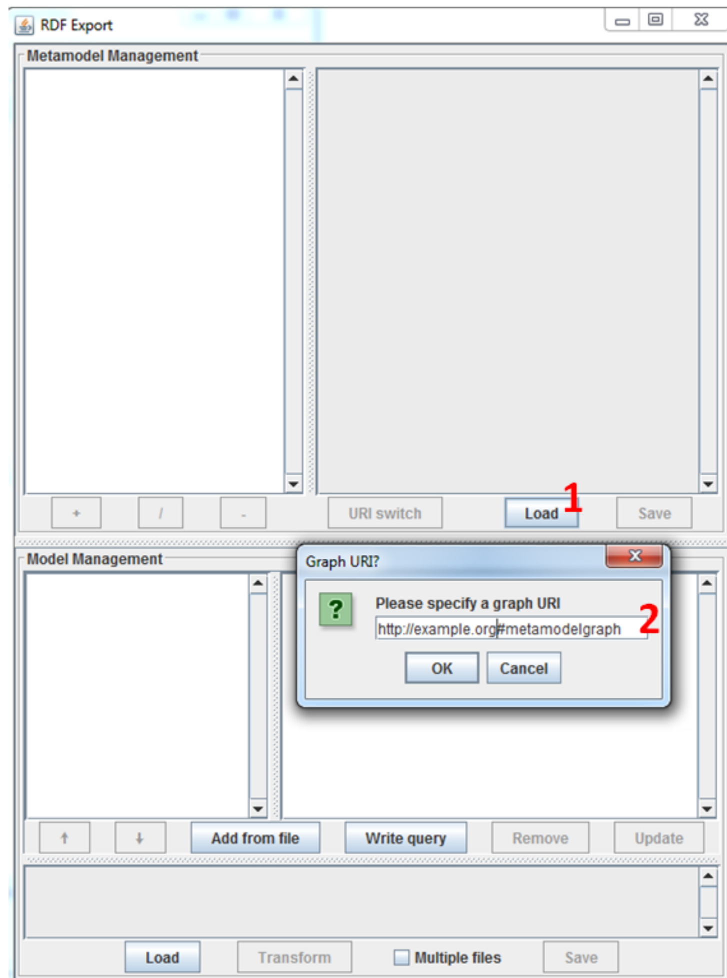


Figura 129 Preluarea metamodelului serializat în RDF Transformer

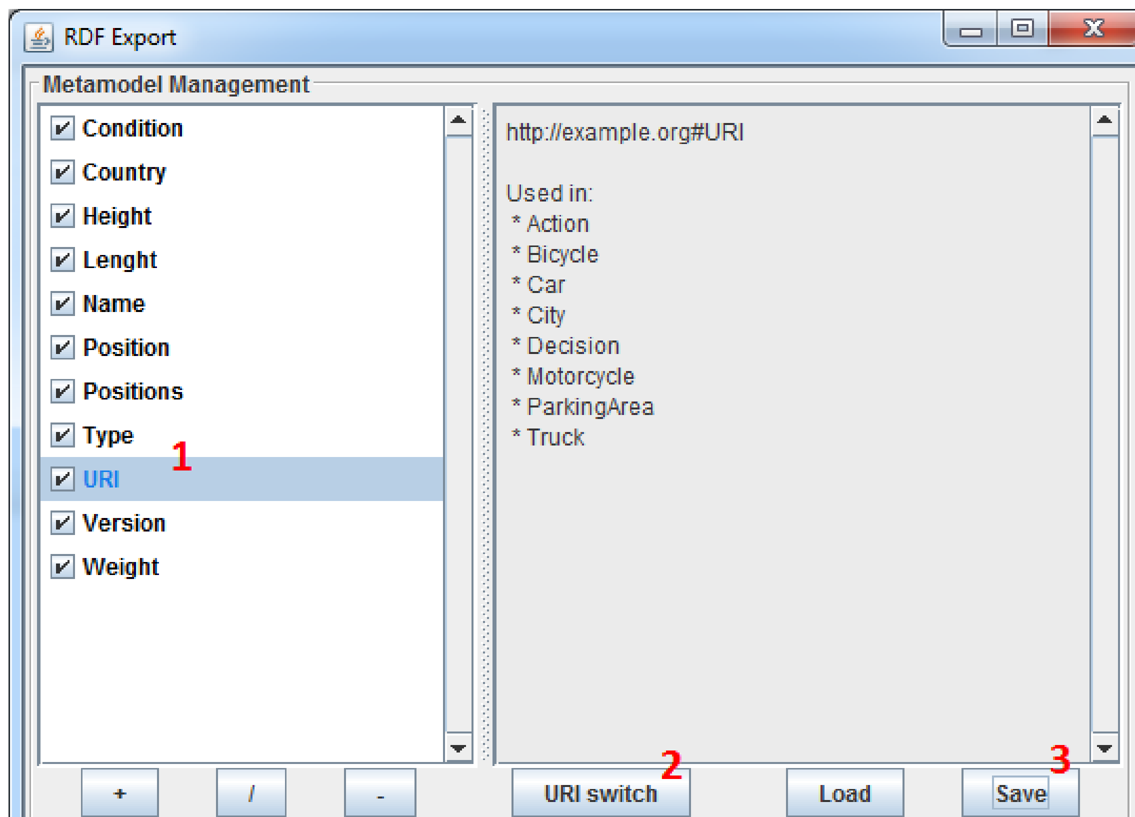


Figura 130 Indicarea atributului care va prelua rolul de identificatory

Pasul 9. Transformarea modelelor în format RDF

Partea de jos a ferestrei va fi folosită pentru a transforma modelele, urmând pașii indicați în Figura 131:

1. Încărcați modelele serializate (*models.xml*);
2. Selectați Transform;
3. Transformarea va solicita să specificăm un URI de bază care să fie aplicat asupra tuturor identificatorilor de la toate elementele. Pentru păstra simple interogările următoare, vom presupune același identificator URI de bază ca cel definit anterior *(http://example.org#)*
4. Rezultatul transformării ar trebui de asemenea salvat ca fișier TriG.

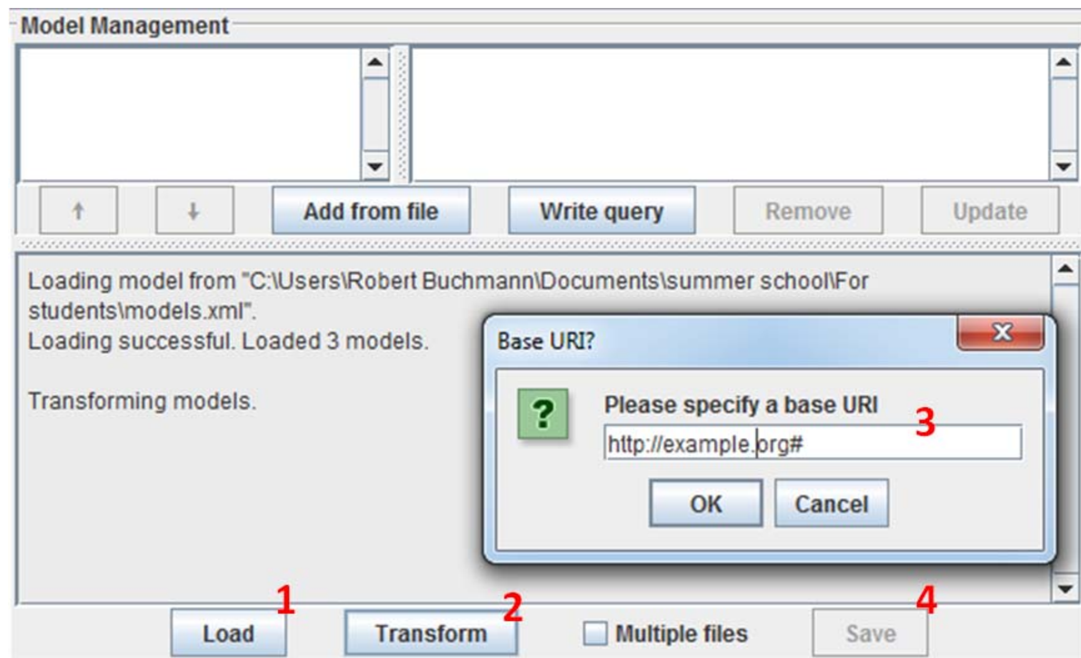


Figura 131 Transformarea informațiilor din modele în Linked Data

Pasul 10. Încărcarea metamodelului, modelelor și a datelor într-un depozit RDF

Pentru acest exercițiu, trebuie să creăm o bază de date RDF4J. Încărcați în acesta cele trei fișiere TriG – metamodelul, modelele și datele din tabel.

Următoarea interogare va extrage numele tuturor curierilor păstrat în modele. De aceea, interogarea se bazează doar pe informația din modele*:*

```
PREFIX :<http://example.org#>

SELECT ?modelname
WHERE
{
?model a :CourierTask; :Name ?modelname
}
```

Rezultatele sunt afișate în Figura 132:

Modelname
"OnDemandTask"
"ScheduledTask"

Figura 132 Rezultatele interogării

Următoarea interogare va extrage identificatorii pentru toate parcările, precum și numărul de locuri libere corespunzătoare, din datele externe:

```
PREFIX :<http://example.org#>

SELECT distinct *
WHERE
{
GRAPH :LegacyData {?park :availability ?spaces}
}
```

Rezultatele sunt afișate în Figura 133:

Park	Spaces
<http://example.org#ParkingA>	2
<http://example.org#ParkingB>	0
<http://example.org#ParkingD>	0
<http://example.org#ParkingC>	4
<http://example.org#ParkingE>	3

Figura 133 Rezultatele interogării

Următoarea interogare va extrage numele orașelor, a parcărilor și a locurilor de parcare libere corespunzătoare, care sunt relevante pentru modelul ”OnDemandTask”. De aceea, această interogare folosește relațiile din modele cât și datele externe.

```
PREFIX :<http://example.org#>
PREFIX cv:<http://www.comvantage.eu/mm#>

SELECT DISTINCT ?cityname ?parkingname ?spaces
WHERE
{
GRAPH :graphmetadata {?taskmodel :Name "OnDemandTask"}

GRAPH ?taskmodel
{?action :requiresParkingInCity ?city. ?city cv:described_in ?parkmodel}
GRAPH ?parkmodel
{?city :contains ?parking.  ?city :Name ?cityname. ?parking :Name ?parkingname}
GRAPH :LegacyData
{?parking :availability ?spaces}
}
```

Rezultatele sunt afișate în Figura 134:

Cityname	Parkingname	Spaces
"Vienna"	"ParkingA"	2
"Vienna"	"ParkingB"	0
"Klagenfurt"	"ParkingC"	4
"Munich"	"ParkingD"	0

Figura 134 Rezultatele interogării

Dezvoltatorii de aplicații pot folosi acum datele ca back-end. Aplicațiile vor putea extrage atât informația din modele, din date oferite de terțe părți, cât și o combinație a acestora prin execuția unor astfel de interogări. Acest exemplu demonstrativ nu va intra în detaliile programării unor astfel de aplicații. Interogarea bazelor de grafuri RDF a fost deja exemplificată în PHP și Python.

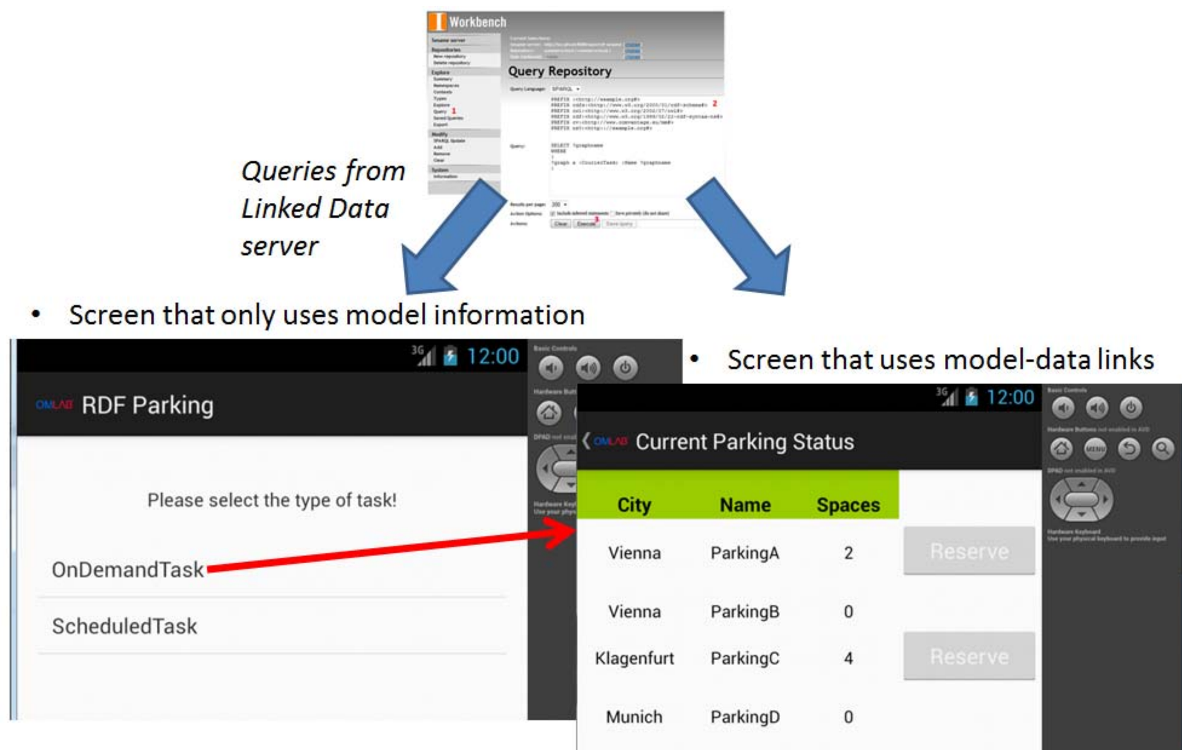


Figura 135 Aplicație pentru dispozitiv mobil ce folosește informații din modele și date externe

8.9 Structura grafurilor exportate

Pentru metamodel, fișierul RDF rezultat va conține un graf numit :metamodelgraph. Exemple de afirmații pe care ați putea să le găsiți în acesta:

```
:metamodelgraph {

:Action
        a       rdfs:Class ;
        rdfs:label "Action" ;
        rdfs:subClassOf cv:Instance_class .
```

În acest fel se declară o clasă ADOxx

```
:requiresParkingInCity
        a       rdf:Property , cv:na_relation_property ;
        rdfs:label "requiresParkingInCity" ;
        cv:used_in :Action .
```

În acest fel se declară o hiperlegătură ADOxx (interref)

```
:Country
        a       rdf:Property , cv:attribute_property ;
        rdfs:label "Country" ;
        cv:used_in :City .
```

În acest fel se declară un atribut ADOxx

```
:contains
        a       rdf:Property , cv:na_relation_property ;
        rdfs:label "contains" .
```

În acest fel se declară o relație ADOxx fără atribute

```
:next
        a       rdfs:Class ;
        rdfs:label "next" ;
        rdfs:subClassOf cv:Relation_class .
```

În acest fel se declară o relație ADOxx cu atribute

```
:URI  a       rdf:Property , cv:attribute_property ;
        rdfs:label "URI" ;
        rdfs:subPropertyOf owl:sameAs ;
        cv:used_in :Truck , :Bicycle , :City , :ParkingArea , :Motorcycle , :Car , :Decision , :Action .
```

În acest fel se declară atributul ce este ales ca fiind identificator

```
cv:used_in a       rdf:Property .
cv:attribute_property a       rdfs:Class ; rdfs:subClassOf rdf:Property .
cv:na_relation_property  a       rdfs:Class ; rdfs:subClassOf rdf:Property .
cv:from_instance  a       rdf:Property .
cv:to_instance  a       rdf:Property .
cv:Instance_class  a       rdfs:Class .
cv:Relation_class  a       rdfs:Class .
cv:Model_class  a       rdfs:Class .
cv:described_in  a       rdf:Property .
```

Acestea sunt elementele vocabularului ce vor fi folosite în model

Acum să vedem cum arată informația din modele. Presupunem că avem următoarele modele exportate:

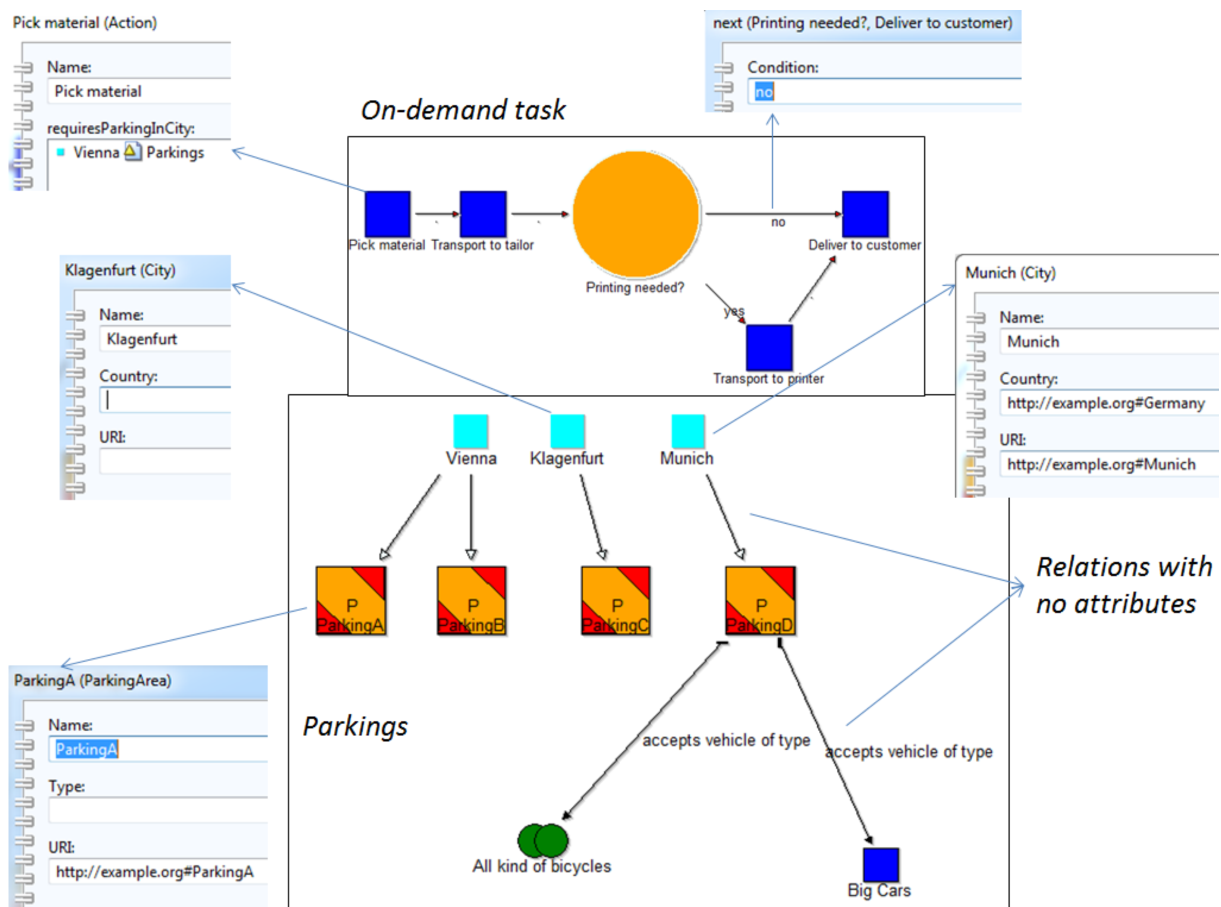


Figura 136 Modelele exportate din instrumentul de modelare

În fișierul exportat avem câte un graf pentru fiecare model exportat și un graf adițional cu metadatele modelului (tipul modelului, numele etc.).

Mai întâi analizăm :graphmetadata:

```
:graphmetadata

{:ParkingMap-Parkings_
        a       cv:Model_class , :ParkingMap ;
        :Name   "Parkings" ;
        :Version "" .

   :CourierTask-OnDemandTask_
        a       :CourierTask , cv:Model_class ;
        :Name   "OnDemandTask" ;
        :Version "" .
}
```

Puteți observa că au fost exportate trei modele. Fiecare dintre acestea are:

- un tip generic cv:Model_class

- un tip specific (tipul de model ADOxx, de ex. :CourierTask, :ParkingMap)
- un nume afișabil dat în instrumentul de modelare (:Name)
- un identificator compus din tipul modelului, numele modelelui și un caracter underscore (:CourierTask-OnDemandTask_)

Apoi analizăm modelul Courier task:

```
:CourierTask-OnDemandTask_

{
 :Action-86165-Pick_material
        a       cv:Instance_class , :Action ;
        :Name   "Pick material" ;
        :Position "NODE x:1.5cm y:3.5cm w:1.16cm h:1.16cm index:2" ;
        :requiresParkingInCity  :City-86124-Vienna .
```

Acesta este exportul tipic care se face pentru obiectele de modelare:

- este de tip :Action (din graful metamodel) și de tip cv:Instance_class (clasa care conține toate obiectele de modelare);
- are un nume afișabil Name;
- este posibil să aibă și alte atribute specificate de ADOxx (care sunt irelevante pentru RDF, în afară de cazul în care se dorește importul modelelor pentru a le afișa în concordanță cu modul de așezare original;
- are o legătură :requiresParkingInCity către Vienna.

De fiecare dată când avem o legătură, destinația legăturii va folosi o declarație cv:described_in :

```
:City-86124-Vienna
        cv:described_in :ParkingMap-Parkings_ .
```

Aceasta indică cărui model aparține Vienna. O interogare ar trebui să caute în acel model informații suplimentare despre acest obiect.

```
<http://example.org#next-86178-Printing_needed?-Deliver_to_customer>
        a       cv:Relation_class , :next ;
        :Condition "no" ;
        :Positions "EDGE 0 index:10" ;
        cv:from_instance <http://example.org#Decision-86162-Printing_needed?> ;
        cv:to_instance :Action-86174-Deliver_to_customer .
```

Acesta este exportul tipic pentru relații cu atribute. Proprietățile sunt:

- este de tip :next (din graful metamodel) și de tip cv:Relation_class (clasa tuturor relațiilor de modelare);
- cv:from_instance indică numele obiectului de unde pornește săgeata (decizia ”Printing needed?”);
- cv:to_instance indică obiectul unde se oprește săgeata (Deliver to customer);
- este posibil să aibă și alte atribute suplimentare (:Condition);

Observați că în unele cazuri operația de export va genera identificatori URI compleți în locul celor prefixați. Acest lucru se întâmplă când unele caractere din URI nu sunt permise în versiunea cu prefixe (semnul interogării).

Vom studia acum modelul parkings:

```
:ParkingMap-Parkings_ {

:City-86124-Vienna
        a       :City , cv:Instance_class ;
        :Name   "Vienna" ;
        :Position "NODE x:4cm y:1.5cm w:.63cm h:.63cm index:1" ;
        :contains :ParkingB , :ParkingA .
```

Acesta este exportul tipic pentru obiectele de modelare (precum Pick material). În plus, aici putem observa că în cazul unei relații care nu are atribute aceasta va deveni o proprietate directă (:City-86124-Vienna :contains :ParkingA). Astfel nu mai este necesară combinația cv:from_instance și cv:to_instance.

```
:ParkingA
    a     :ParkingArea , cv:Instance_class ;
    :Name   "ParkingA" ;
    :Position "NODE x:1.5cm y:5cm w:1.41cm h:1.41cm index:4" ;
    :URI    :ParkingA .
```

Și acesta este un obiect de modelare. Ceea ce este nou în acest caz este faptul că identificatorul :ParkingA este folosit în locul celui complicat care ar fi generat (precum :City-86124-Viena). Putem să suprascriem valoarea URI exportată cu ajutorul atributului URI. Aceasta poate fi realizată pentru orice obiect (deoarece toate au atributul URI) și același lucru poate fi făcut și cu valorile atributelor de tip string:

```
  <http://expl.at#Munich>
    a     :City , cv:Instance_class ;
    :Country <http://expl.at#Germany> ;
    :Name   "Munich" ;
    :Position "NODE x:13.5cm y:1.5cm w:.63cm h:.63cm index:3" ;
    :URI    <http://expl.at#Munich> ;
    :contains :ParkingD.
```

În acest caz, valoarea de la :Country primește un URI în locul unui simplu string, iar identificatorul de la Munich de asemenea, a fost înlocuit cu ceea ce a fost oferit în model (a se vedea imaginea cu modele).

```
  :Action-86165-Pick_material
    :requiresParkingInCity
        :City-86124-Vienna ;
    cv:described_in :CourierTask-OnDemandTask_ .
```

Acestea sunt câteva afirmații legate de acțiunea Pick material. De ce apar în acest graf? Datorită faptului că pentru orice legătură dintre modele (inter-model) legătura este salvată atât în graful sursă cât și cel destinație. Și în fiecare sin acestea o relație cv:described_in va indica unde am putea să căutăm celălalt capăt al legăturii.

Presupunând că toate informațiile legate de proiectul cu parcări sunt încărcate în RDF4J (după cum sunt descrise în ghidul demonstrativ) încercați următoarele interogări:

Lista cu toate modelele:

```
select ?x
where
{?x a cv:Model_class}
```

Lista cu toate obiectele:

```
select ?x
where
{?x a cv:Instance_class}
```

Lista cu toate obiectele din modelul ”Parkings”:

```
select ?x
where
{
graph :graphmetadata {?g :Name "Parkings"}
graph ?g {?x a cv:Instance_class}
}
```

Am putea obține această informație și dacă folosim direct identificatorul modelului (însă numele este mai ușor de reținut):

```
select ?x
where
{
graph :ParkingMap-Parkings_ {?x a cv:Instance_class}
}
```

Lista cu toate atributele și unde sunt acestea folosite:

```
select ?x ?y
where
{
?x a cv:attribute_property; cv:used_in ?y
}
```

Lista cu toate atributele disponibile pentru relația Next:

```
select distinct ?p
where
{
?s a :next; ?p ?o.
?p a cv:attribute_property;
}
```

Lista cu toate atributele disponibile pentru cities:

```
select distinct ?p
where
{
?s a :City; ?p ?o.
?p a cv:attribute_property;
}
```

Lista cu toate relațiile fără atribute (inclusiv legăturile):

```
select distinct ?p
where
{
?p a cv:na_relation_property
}
```

Lista cu toate legăturile:

```
select distinct ?p
where
{
?p a cv:na_relation_property.
?s ?p ?o.
?o cv:described_in ?m
}
```

Lista cu toate legăturile, incluzând clasa sursă a acestora:

```
select distinct ?p ?t
where
{
?p a cv:na_relation_property.
?s ?p ?o.
?o cv:described_in ?m.
?s a ?t
}
```

Lista cu relații fără atribute (fără legături):

```
select distinct ?p
where
{
?p a cv:na_relation_property.
minus
{?s ?p ?o.
?o cv:described_in ?m}
}
```

Lista cu relații cu atribute:

```
select distinct ?p
where
{
?p rdfs:subClassOf cv:Relation_class.
}
```

Observați că relațiile cu atribute sunt noduri RDF, în timp ce relațiile fără atribute (inclusiv legăturile) sunt predicate RDF! Aceasta se datorează faptului că la relațiile cu atribute, fiecare apariție a relației poate să aibă valori diferite pentru atribute, de aceea trebuie să fie tratată ca un obiect distinct!

Lista cu toate relațiile next împreună cu numele surselor și a destinațiilor:

```
select distinct ?p ?nx ?ny
where
{
?p a :next; cv:from_instance ?x; cv:to_instance ?y.
?x :Name ?nx.
?y :Name ?ny
}
```

Lista cu toate asocierile dintre acțiuni și orașe:

```
select distinct ?nx ?ny
where
{
?x :requiresParkingInCity ?y.
?x :Name ?nx.
?y :Name ?ny
}
```

Lista cu toate asocierile dintre acțiuni și zonele de parcare:

```
select distinct ?nx ?nz
where
{
?x :requiresParkingInCity ?y.
?y :contains ?z.
?x :Name ?nx.
?z :Name ?nz
}
```

Lista modelelor care conțin legături către orașe:

```
select distinct ?m
where
{
?x :requiresParkingInCity ?y.
?x cv:described_in ?m
}
```

Lista cu toate modelele care conțin legături către modelul ”Parkings”:

```
select distinct ?m
where
{
graph ?g
{?x :requiresParkingInCity ?y.
?x cv:described_in ?m}
graph :graphmetadata
{?g :Name "Parkings"}
}
```

Lista cu noduri (numele) care urmează imediat după decizia ”Printing needed?”:

```
 select distinct ?ny
where
{
?x :Name "Printing needed?".
?rel cv:from_instance ?x; cv:to_instance ?y.
?y :Name ?ny.
```

```
}
```

Nodul (numele) care urmează imediat după decizia ”Printing needed?” dacă răspunsul este ”nu”:

```
select distinct ?ny
where
{
?x :Name "Printing needed?".
?rel cv:from_instance ?x; cv:to_instance ?y; :Condition "no".
?y :Name ?ny.
}
```

Nodul (numele) care precedă nodul ”Transport to tailor”:

```
select distinct ?ny
where
{
?x :Name "Transport to tailor".
?rel cv:from_instance ?y; cv:to_instance ?x.
?y :Name ?ny.
}
```

Generează o relație directă :followedBy pentru fiecare apariție a relației :next

```
insert
{?x :followedBy ?y}
where
{?rel a :next; cv:from_instance ?x; cv:to_instance ?y}
```

(Folosiți secțiunea SPARQL Update!)

Lista cu toate nodurile care urmează după ”Transport to tailor”:

```
select distinct ?ny
where
{
?x :Name "Transport to tailor".
?x :followedBy+  ?y.
?y :Name ?ny
}
```

Lista cu toate locurile de parcare care acceptă biciclete:

```
select distinct ?x
where
{
?x a :ParkingArea; :acceptsVehiclesOfType ?y.
?y a :Bicycle.
}
```

Lista cu toate zonele de parcare din Germania (presupunând că știm identificatorul public http://expl.at#Germany)

```
select distinct ?x
where
{
?x a :ParkingArea.
?y :contains ?x; :Country <http://expl.at#Germany>.
}
```

Lista cu toate orașele unde trebuie să se meargă după acțiunea ”Transport to tailor”:

```
select distinct ?nc
where
{
?x :Name "Transport to tailor".
?x :followedBy+  ?y.
?y :requiresParkingInCity ?c.
?c :Name ?nc
}
```

Acum că avem toată informația din modele pentru a fi interogată ca Linked-Data, putem mai departe să o legăm de orice sursă de date. Presupunem că avem următoarele date disponibile (fișierul data.trig):

```
@prefix : <http://example.org#>.
@prefix c: <http://cities.org#>.

:LegacyData
{
:ParkingA :availability 2 ; a :ParkingArea.
:ParkingB :availability 0 ; a :ParkingArea.
:ParkingC :availability 4 ; a :ParkingArea.
:ParkingD :availability 0 ; a :ParkingArea.
:ParkingE :availability 3 ; a :ParkingArea.
c:Vienna c:population "1.7 mil".
c:Munich c:population "1.5 mil".
c:Klagenfurt c:population "100 thou".
}
```

Încărcați-l în cadrul aceluiași depozit.

Acum putem executa interogări care să beneficieze de conexiunile dintre modele și date. Următoarea interogare va afișa acțiunile, zonele de parcare necesare pentru acele acțiuni și disponibilitatea pentru fiecare zonă de parcare:

```
select distinct ?nx ?np ?y
where
{
?x :requiresParkingInCity/:contains ?p.
?p :availability ?y.
?x :Name ?nx.
?p :Name ?np.
}
```

Conexiunea a fost posibilă deoarece am refolosit aceeași identificatori pentru zonele de parcare în ambele modele (prin intermediul atributului URI) precum și în datele externe. Totuși acest lucru nu este întotdeauna posibil (câteodată modele sunt create înaintea datelor). De exemplu, orașele nu ar avea aceeași identificatori.

În astfel de cazuri, putem crea o uniune prin declararea echivalențelor. Creați următoarele asocieri (fișierul *citymappings.trig)* și încărcați-le în RDF4J:[77]

```
@prefix : <http://example.org#>.
@prefix c: <http://cities.org#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

:citymappings
{
c:Vienna owl:sameAs :City-86124-Vienna.
c:Munich owl:sameAs <http://expl.at#Munich>.
c:Klagenfurt owl:sameAs :City-86127-Klagenfurt.
}
```

Acum putem folosi aceste asocieri pentru a conecta informația din modele cu informații despre populație.

Următoarea interogare va extrage populația din toate orașele în care sarcinile modelate solicită parcare:

```
select distinct ?nx ?c1 ?pop
where
```

[77] numărul care a fost generat în timpul exportului în cadrul fiecărui URI (de ex. 86124) poate să difere în instrumentul de modelare. De aceea, prin suprascrierea acestuia cu proprietatea URI este mai ușor să legăm elementele modelului cu identificatorii URI existenți.

```
{
?x :requiresParkingInCity ?c1; :Name ?nx.
graph :citymappings {?c2 owl:sameAs ?c1}
graph :LegacyData {?c2 c:population ?pop}
}
```

Toate aceste interogări se pot integra într-un mediu de programare prin librării capabile să proceseze grafuri și să se conecteze la interfața REST a serverului RDF4J. S-au exemplificat în acest sens instrumentele EasyRDF pentru PHP, rdflib, SparqlWrapper și urllib pentru Python.

Documentația standardelor utilizate

JSON - https://www.json.org/,

JSON Schema - http://json-schema.org/,

JSON-LD - https://json-ld.org/,

UML – Unified Modeling Language http://www.uml.org/

BPMN - Business Process Model and Notation http://www.bpmn.org/

MOF – Meta Object Facility http://www.omg.org/mof/

Archimate – http://pubs.opengroup.org/architecture/archimate3-doc/toc.html,

HTML Microdata - https://www.w3.org/TR/microdata/,

OWL Web Ontology Language https://www.w3.org/OWL/,

RDF N-Triples https://www.w3.org/TR/n-triples/

RDF - Resource Description Framework - https://www.w3.org/RDF/,

RDF Schema https://www.w3.org/TR/rdf-schema/,

RDF TriG https://www.w3.org/TR/trig/,

RDF Turtle https://www.w3.org/TR/turtle/

RDFa https://www.w3.org/TR/rdfa-primer/,

SPARQL Query Language for RDF https://www.w3.org/TR/rdf-sparql-query/

XML Extensible Markup Language - https://www.w3.org/XML/

XML Schema https://www.w3.org/standards/xml/schema

XPath https://www.w3.org/TR/xpath/

XSLT - https://www.w3.org/TR/xslt/

Paginile oficiale ale instrumentelor utilizate în această carte sunt indicate prin note de subsol în exercițiile în care sunt utilizate.

Bibliografie științifică

Resurse documentare din literatura științifică în care s-au pus bazele tehnologiilor semantice prezentate în această carte:

1. Fill, H. G., Redmond, T., & Karagiannis, D. (2012). Formalizing meta models with FDMM: the ADOxx case. In International Conference on Enterprise Information Systems (pp. 429-451). Springer, Berlin, Heidelberg.
2. Heath, T., & Bizer, C. (2011). Linked data: Evolving the web into a global data space. Synthesis lectures on the semantic web: theory and technology, 1(1), 1-136.
3. Karagiannis, D. (2015). Agile modeling method engineering. In Proceedings of the 19th Panhellenic Conference on Informatics (pp. 5-10). ACM.
4. Karagiannis, D., Mayr, H. C., & Mylopoulos, J. (2016). Domain-Specific Conceptual Modeling. Springer International Publishing.
5. Staab, S., & Studer, R. (Eds.). (2010). Handbook on ontologies. Springer Science & Business Media.
6. Staab, S., Walter, T., Gröner, G., & Parreiras, F. S. (2010). Model driven engineering with ontology technologies. In Reasoning web. semantic technologies for software engineering (pp. 62-98). Springer, Berlin, Heidelberg.
7. Van der Aalst, W. M. (2009). Process-aware information systems: Lessons to be learned from process mining. In Transactions on petri nets and other models of concurrency II (pp. 1-26). Springer Berlin Heidelberg.
8. Prezentările Școlii de Vară Internaționale Next Generation Enterprise Modelling (2014-2018) http://nemo.omilab.org/2018/

Resurse documentare din literatura științfică unde au fost aplicate tehnologiile și ideile sugerate în această carte:

1. Buchmann R. A., Cinpoeru M., Harkai A., Karagiannis, D. (2018) "Model-Aware Software Engineering - A Knowledge-based Approach to Model-Driven Software Engineering", In Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018), March 23-24, Funchal, Madeira, Portugal, 2018 (pp. 233-240), ISBN: 978-989-758-300-1.
2. Buchmann R. A., Ghiran A. M., (2017) "Engineering the Cooking Recipe Modelling Method: a Teaching Experience Report", In Proceedings of the 1st International Workshop on Practicing Open Enterprise Modeling within OMiLAB (PrOse2017), November 22-24, 2017, Leuven, Belgium.
3. Buchmann R. A., Ghiran A. M., Osman C. C., Karagiannis D. (2018) "Streamlining Semantics from Requirements to Implementation Through Agile Mind Mapping Methods", In: Kamsties E., Horkoff J., Dalpiaz F. (eds) Requirements Engineering: Foundation for Software Quality. REFSQ 2018. Lecture Notes in Computer Science, vol 10753. Springer, (pp. 335-351), ISBN 978-3-319-77243-1.
4. Buchmann, R.A., Karagiannis, D. (2016). Enriching linked data with semantics from domain-specific diagrammatic models. Business & Information Systems Engineering, 58(5), 341-353.
5. Ghiran, A. M., Buchmann, R. A., Osman, C. C., Karagiannis, D. (2017). Streamlining Structured Data Markup and Agile Modelling Methods. In IFIP Working Conference on The Practice of Enterprise Modeling (pp. 331-340). Springer.
6. Harkai A., Cinpoeru M., Buchmann R. A. (2018) "Repurposing Zachman Framework Principles for Enterprise Model-Driven Engineering.", In Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018), - Volume 2, March 21-24, Funchal, Madeira, Portugal, 2018, (pp. 682-689), ISBN: 978-989-758-298-1.
7. Karagiannis D., Buchmann R. A. (2018) "A Proposal for Deploying Hybrid Knowledge Bases: the ADOxx-to-GraphDB Interoperability Case", In Proceedings of the 51st Hawaii International Conference on System Sciences (HICSS), January 3-6, 2018, Hawaii, USA, (pp. 4055-4064), ISBN 978-0-9981331-1-9
8. Karagiannis D., Buchmann R. A., Walch M. (2017) "How Can Diagramatic Conceptual Modelling Support Knowledge Management?". In Proceedings of the 25th European Conference on

Information Systems (ECIS), Guimarães, Portugal, June 5-10, 2017, (pp. 1568-1583). ISBN 978-989-20-7655-3.



ISBN: 978-606-37-0374-4